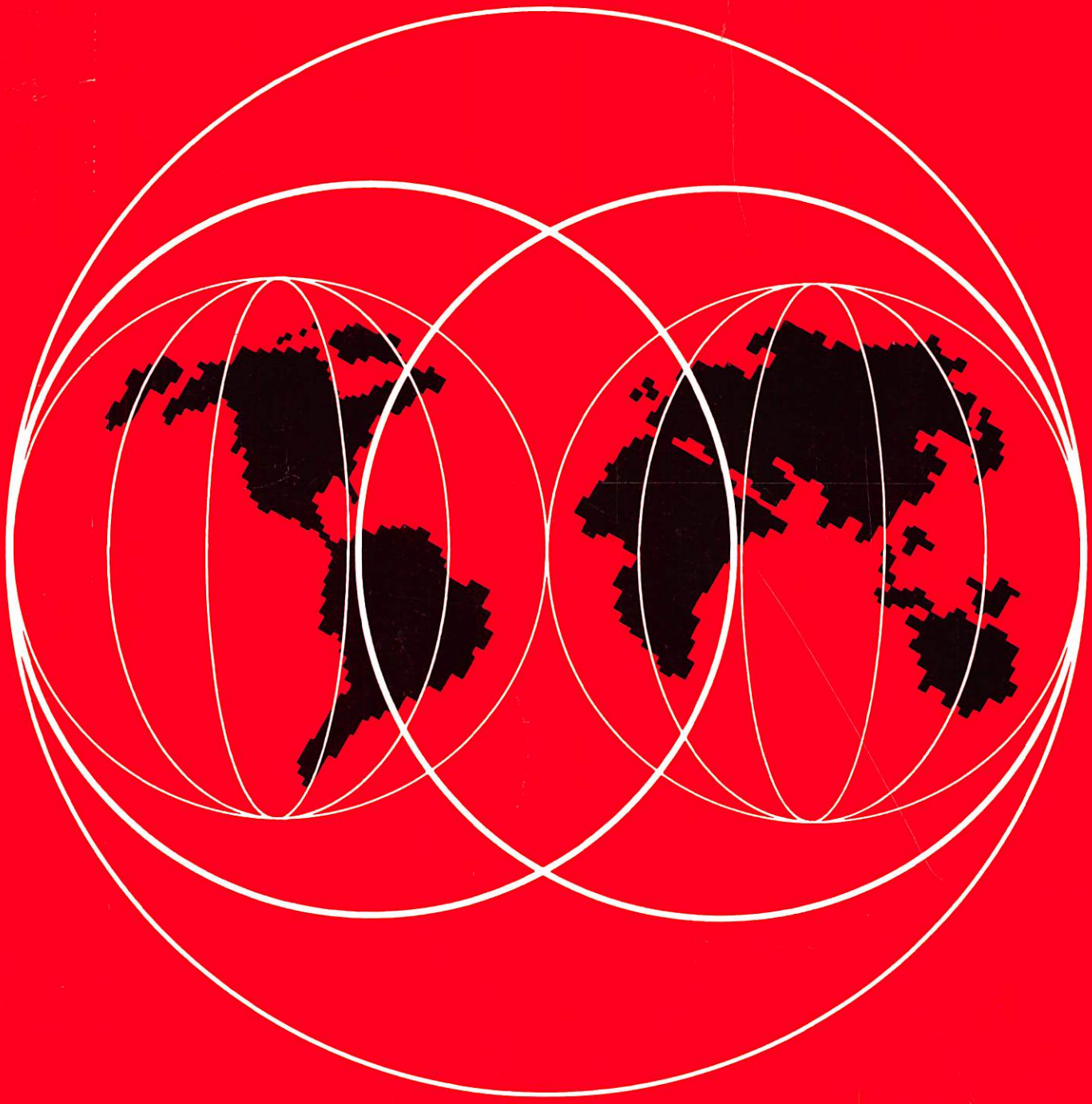

IBM

International Technical Support Centers

OS/2 Version 2

Volume 4: Writing Applications

GG24-3774-01

OS/2 Version 2
Volume 4: Writing Applications

Document Number GG24-3774-01

January 1993

International Technical Support Center
Boca Raton

Take Note!

Before using this information and the product it supports, be sure to read the general information under "Special Notices" on page xxi.

Second Edition (January 1993)

This edition applies to Version 2.0 of Operating System/2.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for reader's comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, International Technical Support Center
Dept. 91J, Building 235-2 Internal Zip 4423
901 NW 51st Street
Boca Raton, Florida 33432

Abstract

This document describes application development for OS/2 Version 2.0. It forms Volume 4 of a five volume set; the other volumes are:

- *OS/2 Version 2.0 - Volume 1: Control Program*, GG24-3730
- *OS/2 Version 2.0 - Volume 2: DOS and Windows Environment*, GG24-3731
- *OS/2 Version 2.0 - Volume 3: Presentation Manager and Workplace Shell*, GG24-3732
- *OS/2 Version 2.0 - Volume 5: Print Subsystem*, GG24-3775

The entire set may be ordered as *OS/2 Version 2.0 Technical Compendium*, GBOF-2254.

This document is intended for IBM system engineers, IBM authorized dealers, IBM customers, and others who require a knowledge of application development under OS/2 Version 2.0.

This document assumes that the reader is generally familiar with the function provided in previous releases of OS/2.

PS (409 pages)

Acknowledgements

The advisors for this project were:

Hans J. Goetz
International Technical Support Center, Boca Raton

Giffin Lorimer
International Technical Support Center, Boca Raton

The authors of this document are:

Alan Chambers
IBM United Kingdom

Franco Federico
IBM United Kingdom

Douglas Pearless
IBM New Zealand

Neil Stokes
IBM Australia

This document was compiled and published with the aid of the International Technical Support Center, Boca Raton.

Thanks to the following people for the invaluable advice and guidance provided in the production of this document:

Ian Ameline
IBM Development Laboratories, Toronto

Mike Cowlishaw
IBM United Kingdom Development Laboratories, Hursley

David Kerr
IBM Programming Center, Boca Raton

Michael Kogan
IBM Programming Center, Boca Raton

Peter Magid
IBM Programming Center, Boca Raton

Greg Milligan
IBM Canada

Larry Raper
IBM Development Laboratories, Austin

Oliver Sims
IBM United Kingdom

Thanks also to the many people, both within and outside IBM, who provided suggestions and guidance, and who reviewed this document prior to publication.

Thanks to the following people for providing excellent tools, used during production of this document:

Dave Hock (CUA Draw)
IBM Cary.

Jürg von Känel (PM Spy)
IBM Yorktown Heights.

Contents

Figures

Tables

Special Notices

This publication is intended to help the customer in the design and implementation of OS/2 Presentation Manager applications under OS/2 Version 2.0, using object-oriented design and programming principles. The information in this publication is not intended as the specification of any programming interfaces that are provided by OS/2 Version 2.0. See the PUBLICATIONS section of the IBM Programming Announcement for OS/2 Version 2.0 for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS.

The information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy completeness.

The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

The following terms, which are denoted by an asterisk (*) in this publication, are trademarks of the International Business Machines Corporation in the United States and/or other countries:

C/2
COBOL/2
Common User Access
CommonView
CUA
DATABASE 2
DB2
DCF
Document Composition Facility
FORTRAN/2
IBM
Macro Assembler/2
Micro Channel
OfficeVision
Operating System/2
OS/2
Personal System/2
Presentation Manager
PS/2
SAA
System/370
Systems Application Architecture
WIN-OS/2
Workplace Shell

The following terms, which are denoted by a double asterisk (* *) in this publication, are trademarks of other companies.

Intel is a trademark of Intel Corporation.
Lotus is a trademark of the Lotus Development Corporation.
Microsoft is a trademark of Microsoft Corporation.
MS-DOS is a registered trademark of Microsoft Corporation.
Smalltalk/V is a trademark of Digitalk Inc.
Windows is a trademark of Microsoft Corporation.
286, 386, 486, SX are trademarks of Intel Corporation.

Preface

This document is intended as a general introduction to the concepts involved in the design and implementation of applications which will execute in the OS/2 Presentation Manager and Workplace Shell environments under OS/2 Version 2.0. It is not intended to be an exhaustive reference on the subject of Presentation Manager programming, and should be used in conjunction with the official IBM product documentation, and other reference books and documents, which are mentioned herein.

It must be stressed that this document is not intended to teach the reader how to program in the "C" language or how to use the Presentation Manager programming interface, nor is it intended to teach the theory of object-oriented programming. Rather, it serves as a guide to the integration of various object-oriented software engineering techniques with the Presentation Manager application model, in order to produce well-structured, easily-maintainable applications which conform to CUA guidelines.

The information given in this document is generally independent of programming language implementations (with certain exceptions noted in the text), and may be used to develop applications in any supported programming language. However, programming syntax examples used in this document are shown using the "C" language, since this language is commonly used for Presentation Manager application development, and most clearly illustrates the structure of the Presentation Manager and Workplace Shell application models.

This document is intended for:

- Application designers, planners and development managers who require an understanding of the application of object-oriented principles to the Presentation Manager environment, and the productivity gains to be achieved from the use of such principles.
- Programmers who wish to understand the structure of Presentation Manager and Workplace Shell applications, and the techniques by which applications may be constructed so as to achieve maximum function, with optimal levels of reusability and maintainability.

The code examples used in this document are available in electronic form via CompuServe** or through a local IBM Support BBS, as package RB3774.ZIP. IBM employees may obtain the code examples from the package GG243774 PACKAGE on OS2TOOLS.

Second Edition

This Second Edition includes programming information relating to application development under OS/2 Version 2.0, and supercedes the ITSC Technical Bulletin *Presentation Manager Application Development*, GG24-3543.

The document is organized as follows:

- *Chapter 1, "Overview"* provides a brief introduction to the topics covered in this document.

  This chapter is recommended for all readers of the document.

- *Chapter 2, "Operating System/2"* provides a brief technical overview of the OS/2 Version 2.0 environment, comparing and contrasting it with the DOS environment and previous versions of OS/2. The major features of OS/2 Version 2.0 are described and their use by applications is discussed.

  This chapter is recommended for those readers who are not familiar with the OS/2 Version 2.0 operating system environment, in order to provide them with a basic understanding of the capabilities of OS/2 Version 2.0.

- *Chapter 3, "Object-Oriented Applications"* explains the basic principles of object-oriented design and programming. The object-oriented approach is compared and contrasted with the traditional procedural approach in terms of a simple application model, before the extension of the object-oriented paradigm into more complex scenarios is discussed. Some suggestions and guidelines are also offered with regard to application design and implementation using the object-oriented approach.

  This chapter is recommended for readers who do not already possess an understanding of the basic principles of object-oriented programming. This knowledge is essential in order to understand the programming guidelines presented later in the document.

- *Chapter 4, "The Presentation Manager Application Model"* describes the Presentation Manager application model, and illustrates the way in which the application model implements the object-oriented principles introduced in Chapter 3, "Object-Oriented Applications."

  This chapter is recommended for all readers of this document, since it explains the basic structure of a Presentation Manager application, and the way in which the Presentation Manager application model facilitates the creation of object-oriented applications.

- *Chapter 5, "The Flat Memory Model"* describes the 32-bit flat memory model implemented in OS/2 Version 2.0, and discusses the programming considerations which arise from the differences between this memory model and the segmented memory model used by previous versions of OS/2.

  This chapter is recommended for all programmers who intend to develop applications under OS/2 Version 2.0.

- *Chapter 6, "Building a Presentation Manager Application"* describes the major programming techniques required to implement a Presentation Manager application, including recommendations and established conventions in areas such as methods of opening and closing windows, displaying dialogs, communication between windows, managing user responsiveness etc. The chapter also discusses certain software engineering techniques which may be used to enhance the level of modularity and optimize the granularity of the resulting application code.

  This chapter is recommended for programmers and development managers who will be working on the implementation of Presentation Manager applications.

- *Chapter 7, "Workplace Shell and the System Object Model"* describes the system object model introduced in OS/2 Version 2.0, and its implementation by the OS/2 Version 2.0 Workplace Shell. The chapter describes the object-oriented application layer provided by the Workplace Shell, and explains how Workplace Shell objects are defined, created and implemented.

  This chapter is recommended for programmers and development managers who wish to create objects for use on the Workplace Shell desktop.

- *Chapter 8, "Direct Manipulation"* explains the implementation of direct manipulation (drag and drop) techniques for carrying out required tasks in the Presentation Manager and Workplace Shell environments. The chapter discusses the use of these techniques both by Presentation Manager windows and by Workplace Shell objects.

  This chapter is recommended for programmers who wish to implement direct manipulation in their Presentation Manager applications or Workplace Shell object classes.

- *Chapter 9, "Presentation Manager Resources"* discusses the concept of Presentation Manager resources. The chapter covers the types of application resources which may be defined in the Presentation Manager environment, their definition and conventions governing their use.

  This chapter is recommended for all programmers who will develop Presentation Manager applications, since resources are used in most if not all applications.

- *Chapter 10, "Multitasking Considerations"* describes the ways in which multiple threads of execution may be used within a Presentation Manager application, in order to isolate long-running application tasks from the user interface and thereby provide greater application responsiveness to the end user.

  This chapter is recommended for programmers and development managers who will be building Presentation Manager applications which carry out lengthy processing tasks, or which require access to remote devices or systems.

- *Chapter 11, "Systems Application Architecture CUA Considerations"* discusses the implementation of various SAA CUA user interface specifications in Presentation Manager applications. The chapter provides coding examples for a number of CUA techniques such as menu bar handling.

  This chapter is recommended for programmers who wish to implement SAA CUA guidelines in their applications.

- *Chapter 12, "Application Migration"* discusses the migration of Presentation Manager applications to OS/2 Version 2.0 from previous versions of OS/2. Differences in implementation are described, along with additional facilities provided by Presentation Manager under OS/2 Version 2.0.

  This chapter is recommended for application developers with Presentation Manager applications written for previous versions of OS/2, which they wish to modify in order to take full advantage of the capabilities of OS/2 Version 2.0.

- *Chapter 13, "Mixing 16-Bit and 32-Bit Application Modules"* describes the way in which 32-bit applications under OS/2 Version 2.0 may make use of existing 16-bit functions and window procedures, along with restrictions and programming considerations to be borne in mind when developing such applications.

  This chapter is recommended for those programmers working in organizations with existing 16-bit runtime libraries or DLLs, and who wish to make use of functions contained within these libraries.

- *Chapter 14, "Compiling and Link Editing an Application"* describes the steps necessary to compile and link edit a Presentation Manager application under OS/2 Version 2.0, including the use of module definition files. and the

creation of dynamic link libraries to contain application code and Presentation Manager resources.

This chapter is recommended for all programmers who will develop Presentation Manager applications, and who wish to understand how to create executable modules and dynamic link libraries.

- *Chapter 15, "Adding Online Help and Documentation"* examines the provision of online, context-sensitive help information for Presentation Manager applications using the IPF provided with Presentation Manager, and the use of this facility to create online documentation.

  This chapter is recommended for application developers who wish to provide online help for their applications, or who wish to develop online documentation and tutorial programs.

- *Chapter 16, "Problem Determination"* describes some simple techniques for problem determination and resolution in the Presentation Manager environment, and discusses some common application problems.

  This chapter is recommended for all application developers involved in testing and debugging Presentation Manager applications.

- *Chapter 17, "Generic Application Modules"* discusses the use of generic routines to perform commonly used functions within a Presentation Manager application, and identifies a number of areas where generic functions may be successfully applied.

  This chapter is recommended for planners and development managers who will manage a number of application developers working on one or more Presentation Manager applications, and who wish to understand the benefits in terms of consistency and productivity which can be achieved through the use of common routines.

- *Chapter 18, "Managing Development"* provides some guidelines for the use of a local area network (LAN) to facilitate centralized control and administration of the workstation-based application development process.

  This chapter is recommended for planners and development managers who will manage a number of application developers working on one or more Presentation Manager applications, and who wish to understand some of the ways in which a distributed development process may be managed and controlled.

The following appendixes are included in this document:

- *Appendix A, "Naming Conventions"* provides some guidelines for naming conventions to be used with symbols, subroutines and variables in the Presentation Manager environment. These guidelines cover the use of Hungarian Notation for such names.

  This chapter is recommended for planners and development managers who wish to implement a standard series of naming conventions for the application development projects under their control.

- *Appendix B, "Application Program Construction"* presents guidelines for the structuring of applications and their component modules in order to achieve the optimum level of modularity and granularity within an application, thus promoting reuse of application code.

  This chapter is recommended for planners and development managers who wish to gain the maximum productivity benefit over a number of Presentation Manager application development projects.

- *Appendix C, "OS/2 Kernel API Functions"* compares the operating system kernel functions provided in OS/2 Version 2.0 with those provided in OS/2 Version 1.3.

  This chapter is recommended for programmers who will be migrating applications from previous versions of OS/2.

- *Appendix D, "Problem Reporting Worksheet"* provides a worksheet which may be used when following the steps given in Chapter 16, "Problem Determination," to provide effective problem documentation which can then be used to reproduce application errors.

  This chapter is recommended for application developers involved in testing and debugging Presentation Manager applications.

Related Publications

The following publications are considered particularly suitable for a more detailed discussion of the topics covered in this document.

Prerequisite Publications

- *IBM OS/2 Version 2.0 Application Design Guide*, 10G6260
- *IBM OS/2 Version 2.0 Control Program Reference*
- *IBM OS/2 Version 2.0 Presentation Manager Reference*
- *IBM OS/2 Version 2.0 Programming Tools Reference.*

Additional Publications

- *OS/2 Version 2.0 - Volume 1: Control Program*, GG24-3730
- *OS/2 Version 2.0 - Volume 2: DOS and Windows Environment*, GG24-3731
- *OS/2 Version 2.0 - Volume 3: Presentation Manager and Workplace Shell*, GG24-3732
- *OS/2 Version 2.0 - Volume 5: Print Subsystem*, GG24-3775
- *OS/2 Version 2.0 Remote Installation and Maintenance*, GG24-3780
- *The Design of OS/2*, Harvey M. Deitel and Michael J. Kogan, Addison Wesley 1992 ISBN 0-201-54889-5 (SC25-4005)
- *Object Oriented Programming: An Evolutionary Approach*, Brad J. Cox, Addison Wesley 1987 ISBN 0-201-10393-1
- *Programmer's Guide to the OS/2 Presentation Manager*, Michael J. Young, Sybex 1989 ISBN 0-89588-569-7
- *Programming the OS/2 Presentation Manager*, Charles Petzold, Microsoft Press 1989 ISBN 1-55615-170-5
- *IBM OS/2 Version 2.0 Technical Library - Procedures Language/2 REXX Reference*, 10G-6268
- *IBM C Set/2 User's Guide*, SC09-1310
- *IBM C Set/2 Migration Guide*, SC09-1369
- *IBM Systems Application Architecture CUA Advanced Guide to User Interface Design*, SC34-4289
- *IBM Systems Application Architecture CUA Advanced Interface Design Reference*, SC34-4290
- *IBM Systems Application Architecture Common Programming Interface Presentation Reference*, SC26-4359.

IBM®

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM International Technical Support Center
Department 91J, Building 235-2
Internal Zip 4423
901 NORTHWEST 51ST STREET
BOCA RATON FL
USA 33431-1328

Fold and Tape **Please do not staple** Fold and Tape

ITSC Technical Bulletin Evaluation RED000

OS/2 Version 2
Volume 4: Writing Applications

Publication No. GG24-3774-01

Your feedback is very important to us to maintain the quality of ITSO redbooks. **Please fill out this questionnaire and return it via one of the following methods:**

- Mail it to the address on the back (postage paid in U.S. only)
- Give it to an IBM marketing representative for mailing
- Fax it to: Your International Access Code + 1 914 432 8246

Please rate on a scale of 1 to 5 the subjects below.
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction ____

Organization of the book	____	Grammar/punctuation/spelling	____
Accuracy of the information	____	Ease of reading and understanding	____
Relevance of the information	____	Ease of finding information	____
Completeness of the information	____	Level of technical detail	____
Value of illustrations	____	Print Quality	____

Please answer the following questions:

a) Are you an employee of IBM or its subsidiaries? Yes____ No____

b) Are you working in the USA? Yes____ No____

c) Was the bulletin published in time for your needs? Yes____ No____

d) Did this bulletin meet your needs? Yes____ No____

If no, please explain:

_____

_____

What other Topics would you like to see in this Bulletin?

_____

_____

What other Technical Bulletins would you like to see published?

_____

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

Name	Address
Company or Organization	
Phone No.	

Chapter 1. Overview

IBM* OS/2* Presentation Manager* is a graphical user interface facility that allows the creation of object-oriented, event-driven applications which conform to IBM Systems Application Architecture* (SAA*) Common User Access* (CUA*) guidelines. Presentation Manager provides an application execution environment under which such applications are executed, and under which they may take full advantage of the advanced capabilities of the OS/2 operating system environment, as well as a system-level mechanism to handle interaction between the application and the user in a consistent and intuitive manner.

The object-based Presentation Manager application model facilitates the use of object-oriented software engineering principles such as data abstraction and encapsulation. The application of these principles enhances application modularity and thereby contributes to increased potential for code reuse and easier application maintenance through containment of change, thereby achieving higher levels of productivity in the areas of application development and maintenance.

This document examines the Presentation Manager execution environment in order to describe the structure and implementation of Presentation Manager applications, and to illustrate the facilities provided by Presentation Manager to support object-oriented techniques. In addition, the document examines the ways in which CUA guidelines may be implemented by Presentation Manager applications within the object-oriented application model. Particular emphasis is given to the use of software engineering principles which facilitate the creation of reusable code for common application services. This is one of the primary concerns of the object-oriented approach to application design, and is also one aspect of the Systems Application Architecture Common Applications ("red layer") component.

The document also discusses the management of workstation-based application development projects. Historically, workstation applications have typically fallen into the systems software category, or have been "one-off" applications and hence have not been subject to the same rules and disciplines imposed upon the traditionally host-based line-of-business applications. However, as the OS/2 environment begins to provide a viable platform for the implementation of workstation-based and cooperative line-of-business applications, typical corporate investments in workstation software are increasing rapidly, and therefore the management and maintenance of these investments must be considered. Some suggestions on the management of the workstation-based development process are given in Chapter 18, "Managing Development."

1.1 User Interface

The Presentation Manager user interface model facilitates an *intuitive* user interface. While people typically approach their work tasks from a "problem-domain" viewpoint, computers tend to adopt an "operator/operand" approach that is inherently alien to the end user. Traditionally, the required translation between approaches has been left to the user, with applications and their user interfaces written to conform to the computer's viewpoint rather than that of the user. This approach has often led to users having difficulty relating to

the technology, with consequently greater amounts of time and money spent in user training.

In recent times, a growing school of thought has emerged which contends that, with the increasing power of computer systems and particularly with the advent of powerful programmable workstations, the responsibility for this interface translation should lie primarily with the application or the computer system rather than with the user. In order to achieve this, user interfaces must be redesigned in order to operate in an object-action, event-driven manner which corresponds with the users' problem domain viewpoint.

Presentation Manager implements such a user interface, and Presentation Manager applications may thus be designed and implemented in such a way as to provide improved user-friendliness and encourage learning by exploration. The details of the Presentation Manager user interface are described in *OS/2 Version 2.0 - Volume 3: Presentation Manager and Workplace Shell*.

Presentation Manager also facilitates *consistency* between applications by handling the interface between user and application at a system level, providing a number of standard constructs which may be exploited by applications. Since these constructs typically appear and behave in the same way regardless of the application under which they are implemented, a user need learn only one set of user interface guidelines to be able to interact with multiple applications. This consistency reduces confusion for users who work with multiple applications, and reduces the need for extensive application training.

The SAA CUA component provides guidelines for the use of these constructs to fulfill particular input/output requirements within an application, such that a level of consistency is achieved not only in the behaviour of the constructs themselves, but also in their relationship to one another and thus in the behaviour of the application as a whole. These guidelines are documented in the *IBM Systems Application Architecture CUA Advanced Guide to User Interface Design*.

1.2 Object-Oriented Applications

Many definitions of the term *object-oriented programming* may be found in various publications and presentations appearing over the last few years. These definitions often differ widely, and have resulted in a great deal of confusion and debate as to the "true" meaning of the term. It may be justifiably argued that there is no such true meaning, and the term *object-oriented* may be used to describe techniques ranging from simple data abstraction to the full inheritance hierarchies implemented by certain object-oriented development tools.

1.2.1 Object-Oriented Design

For the purpose of discussion within this document, an object-oriented application will be defined as one where **data objects** are the focus of the application. A data object is defined to be a particular representation of a logical data entity. For example, a document being edited may exist in two places: as an image in memory and as a file on a fixed disk. Each of these two forms constitutes a separate data object.

The procedures that operate upon these data objects in order to carry out application functions are encapsulated with the data objects to form **application**

objects. Application objects are logically independent units comprising both data and function, which communicate with one another to request actions, conveyed in the form of **messages** passed between the communicating objects. In object-oriented terminology, the procedures that are invoked to carry out the required actions are known as **methods**.

Several rules apply to the design and behaviour of application objects. These are listed below:

- A data object should be accessible *only* from within a single application object which "owns" the data object. The definition, creation and/or establishment of access to the data object should also be achieved from within the application object; this is known as the principle of **encapsulation**.
- The behaviour of and output from an application object should depend upon, and only upon, the type and contents of the messages it receives. The behaviour of an object should not depend upon any other external source.

  As a corollary to the foregoing principle, the result of passing a particular type of message may also vary, depending upon the type of application object to which it is passed, and that object's interpretation of the message. Adherence to this rule allows the behaviour of an object to differ, depending upon the nature of the messages received by that object; this differing behaviour is known as **polymorphism**.

For ease of processing, application objects with similar properties are grouped into **object classes**. Each object in a class is said to be an **instance** of that class. Application objects within the same class share properties such as data object definitions, class-specific variable definitions and values, and methods. Objects therefore take on the properties of their class; this is known as **inheritance**.

It is the concept of inheritance that provides a distinguishing factor between the two major schools of thought which exist under the object-oriented paradigm:

- The basic precept of the **class-based** theory of object-oriented design is that objects are defined in terms of their class, and that new classes are defined in terms of existing classes, with certain additions and modifications which distinguish the new class. Thus there is a measure of interdependence between object classes, and an **inheritance hierarchy** is formed.

  The primary advantage of the class-based approach is that it eases the task of defining object classes, since each new class belongs to a hierarchy of previously defined classes with their own properties and methods. The application developer therefore need only explicitly define the distinguishing characteristics of each class.

  The major disadvatange of the class-based approach is the consequent high level of interdependence between objects. Since the unit of modularity is the entire inheritance hierarchy, rather than the individual object, reuse of a particular object presupposes reuse of all those objects in its hierarchy upon which the definition of the required object depends.

  The class-based approach therefore provides a high initial productivity to the application developer, although with a consequent reduction in the level of granularity and an increase in run-time overhead.
- The **module-based** theory of application development contends that while objects are defined in terms of their class, each new class is totally defined in its own right, and is not dependent upon the definitions of other classes. Hence there is no inheritance hierarchy under the module-based approach.

The primary advantage of the module-based approach is that it avoids the object interdependence associated with the class-based approach, since each object class contains its own complete definition of properties and methods. Thus the unit of modularity is the individual application object.

The disadvantage of this approach lies in the fact that the application developer is required to define each object class in its entirety, and typically cannot rely on previous definitions.[1] The module-based approach therefore attains a higher level of modularity and independence between application objects, but at the expense of higher initial development time.

The object-oriented approach to application design is most suited to applications where the data is the focus of the application, and is less suitable where the procedure or sequence of actions is the critical factor in the design. However, in mixed situations where only certain parts of an application or application system are procedurally oriented, as is the case with many work tasks, and where the provision of an event-driven user interface is desirable, the object-oriented paradigm can be extended to encompass procedurally oriented tasks. This is discussed further in Chapter 3, "Object-Oriented Applications."

While object-oriented applications deal primarily with the manipulation of data entities and their logical representations, there are many situations where an application must deal with other entities such as remote devices or systems. Administrative procedures defined by or imposed upon an organization may also be viewed as logical entities with which an application must interact. The incorporation of such entities into the object-oriented application paradigm requires an expansion of the concept of an application object to include the definition of and methods pertaining to any type of entity addressed by the application. This broadened definition is fundamental in making the object-oriented application model applicable to virtually any application scenario.

1.2.2 Object-Action Interfaces

For the purpose of discussion within this document, an object-oriented application will also be defined as one that implements an event-driven, object-action user interface such as that specified in the *IBM Systems Application Architecture CUA Advanced Guide to User Interface Design*. With such an interface, a user first selects an object to manipulate, then selects one or more of a defined set of actions to be performed upon that object. The sequence of these actions, and hence the sequence of the dialog with the user, is controlled by the user rather than by the application, where this is possible within the requirements of the work task being performed.

The concepts of object-oriented design and an object-action user interface are distinct but complementary. While it is possible to design and create an object-oriented application without an object-oriented user interface, it it far more difficult to implement a truly event-driven, object-action style of user interface without embracing, at least a certain degree, the object-oriented approach to design and implementation. It thus follows that the SAA CUA user interface model cannot be fully implemented without some measure of adherence to object-oriented design principles. It is the provision of an intuitive, event-driven user interface that constitutes one of the great strengths of the object-oriented paradigm.

---

[1] This may be overcome to some extent through subclassing, which is explained later in this chapter.

1.2.3 Benefits of the Object-Oriented Approach

A fundamental benefit of an object-oriented approach from the viewpoint of the end user is the ability for an application to behave in a manner that parallels a typical human being's natural approach to problem solving. The flexibility of the object-action interface allows scope for individual variation in the approach to a particular work task. However, such a user interface relies upon an object-oriented application implementation in order to allow such flexibility. Such an implementation is in turn dependent upon the correct design approach, which must begin with a focus upon the entities that affect the application, rather than upon the procedures to be performed upon those entities.

The object-oriented paradigm also encourages the concept of data abstraction and encapsulation, whereby the definition of and establishment of access to data objects is achieved from within the application object. Ideally, all access to and manipulation of a data object is carried out from within a single application object, thereby facilitating change management and application maintenance.

Another great benefit of the object-oriented approach is the increased potential for creation of reusable code. The independent nature of application objects enables them to be coupled together in various ways to achieve desired results, with the internal implementation details of each object and its data structures being isolated from the other objects with which it communicates. Applications that manipulate existing data objects may therefore be assembled from a number of existing application objects, thus reducing the time and effort required to develop the application.

This potential for object reuse has also given rise to one of the great criticisms levelled at the object-oriented approach; the "myth" of the completely generic object. Due to the impracticability of foreseeing all possible actions that might be performed on a data object, it is impossible to produce a complete set of methods for that object. Hence an application object might require modification at some stage in its life cycle, and is not truly reusable.

The object-oriented approach overcomes this potential problem by the use of a concept known as **subclassing**, whereby a new application object is created comprised of a data object and a number of new or modified methods which act upon that object. Messages destined for the original application object are diverted to the new object; the original object is said to have been *subclassed*. If the message is of a type with which the new object is explicitly concerned, it processes the message using its own methods. If not, it passes the message on to the original object for processing. In the subclassing process, neither the sending object nor the original receiving object should be aware that subclassing has taken place. Subclassing therefore provides a transparent means for modifying or adding to the behaviour of an existing application object without modifying the object itself.

The general principles of object-oriented design and programming, as they apply to the Presentation Manager environment, are explored more fully in Chapter 3, "Object-Oriented Applications."

1.3 Presentation Manager Application Model

Contrary to popular belief, Presentation Manager provides far more than merely the ability to achieve a windowed, graphical user interface for the display of information on the screen. Rather, Presentation Manager provides a message-driven, object-based execution environment that facilitates the provision of an event-driven, object-action user interface, and supports the implementation of object-oriented design techniques.

Presentation Manager enables the implementation of an object-action user interface by providing an application programming interface that conforms to the guidelines laid down in the *IBM Systems Application Architecture Common Programming Interface Presentation Reference*, and a set of underlying system services that support an object-oriented, event-driven application model. The Presentation Manager programming interface provides user interface constructs which conform to CUA guidelines. However, it must be strongly emphasized that the term "presentation interface" is a misnomer, since Presentation Manager is concerned with far more than merely the display of information on the screen.

The Presentation Manager application model is centered around the concept of **windows**. While a window typically appears as a rectangular area on the screen, it is in fact a "handle" to a data object; a window concerned with data being displayed on the screen is termed a **display window**, whereas a window concerned with an internal data object is known as an **object window**. Each window belongs to a **window class** and is associated with a **window procedure**, which contains the definition of the window's data object(s) and also contains methods to perform all of the processing associated with that window. Windows and their associated window procedures communicate with the user and with each other, by the use of messages that are routed to the appropriate window by Presentation Manager.

Since a window procedure may contain all the processing related to a particular data object, along with the definition of and establishment of access to that data object, Presentation Manager provides a suitable environment for data encapsulation and abstraction, in that the internal representation and workings of a data object may be "hidden" within the window procedure that "owns" the data object. Provided the external interfaces (that is, the message formats) of the window procedure remain unchanged, other window procedures within an application are insulated from changes to the data object or its processing. This provides a powerful tool for the enhancement of application modularity and the containment of change within an application. This in turn facilitates the task of application maintenance and change management, since affected application modules may be easily identified.

A close corellation may be drawn between the concept of an application object and that of a window under Presentation Manager. The window becomes the identity of, or "handle" to an object; the data referenced by the window (whether a display space on the screen or a data file) becomes a data object; the window procedure associated with a window contains the methods to act upon that data object; and actions to be performed by the application object on its data object are conveyed by way of messages routed to the window by Presentation Manager. Although Presentation Manager provides window classes to allow grouping of objects with similar characteristics, a full inheritance hierarchy is not supported, and thus Presentation Manager conforms more closely to the

module-based theory of object-oriented design than to the class-based approach. Development tools such as Smalltalk V/PM** are available to extend the Presentation Manager application model and facilitate implementation of a full inheritance hierarchy.

The Presentation Manager application model, along with the underlying OS/2 environment, affords the ability to store an application object (that is, a data object definition, along with the methods associated with that data object, contained within a window procedure) in a library that may be dynamically linked with an application. This in turn provides the potential to develop and implement applications composed of one or more generic objects linked by a custom-built application harness, which allows applications to be assembled more quickly and at less cost.

The Presentation Manager programming interface includes a mechanism for **subclassing** a window, whereby messages destined for a particular window may be transparently diverted to another window for specialized processing. Implementing an application using generic objects with subclassing to provide specialized methods may greatly reduce the amount of coding required, and consequently reduce the development time and cost of applications.

The general implementation of and support for object-oriented programming principles under Presentation Manager is discussed further in Chapter 4, "The Presentation Manager Application Model." The subject is examined in more detail, and specific examples are discussed, in Chapter 6, "Building a Presentation Manager Application."

1.3.1 Systems Application Architecture Conformance

While Presentation Manager supports and facilitates the implementation of object-oriented design techniques and provides support for the user interface constructs and guidelines laid down by the CUA component of Systems Application Architecture, it does not force an application developer to conform to object-oriented design principles or CUA-conforming user interface guidelines. While the rich function set provided by the Presentation Manager programming interface allows an application developer to interpret and implement CUA guidelines in a number of ways, there are emerging conventions with regard to the implementation of these guidelines.

In order to achieve the benefits which accrue from adherence to object-oriented and CUA principles, a measure of discipline is required on the part of the application developer, so as to implement the application in such a way that the maximum degree of object-independence and reusability is attained, and that the optimal level of conformance to CUA conventions is achieved. The subject of CUA conformance is discussed in detail in Chapter 11, "Systems Application Architecture CUA Considerations."

Note that CUA conformance, along with consistency in the implementation of application functions and user interface constructs, may be enforced or enhanced through the use of standard functions and subroutines contained in code libraries. The creation of such libraries is facilitated by the modular nature of the Presentation Manager environment, and by the dynamic linking capabilities of the OS/2 operating system. This subject is discussed further in Chapter 17, "Generic Application Modules."

1.3.2 Online Help and Documentation

Presentation Manager also supports the development of online, context-sensitive help panels, along with online documents for support of applications, business processes and computer-based training. Such information may be displayed in windows on the Presentation Manager desktop, using the **Information Presentation Facility (IPF)**, which is shipped with the operating system.

Help panels displayed using IPF are context-sensitive, thereby allowing the user to request help on a specific topic, and the application to that help in a window on the Presentation Manager desktop. Help panels within an application may be indexed, which allows a user to search for help on related topics in addition to the topic initially requested.

Phrases or illustrations within panels may be marked as selectable, and used to display additional information, initiate application events or start new applications. This capability is provided by the **hypertext** and **hypergraphics** facilities of IPF.

Online documents may also be generated by IPF. Such documents are not linked to applications; they act as applications in their own right, and indeed may be used to initiate the execution of application programs using the hypertext facility of IPF. Online documents may also be indexed, and keyword searches may be conducted on document files; these facilities are part of IPF.

Information Presentation Facility is described in detail in Chapter 15, "Adding Online Help and Documentation."

1.4 The Workplace Shell

Under OS/2 Version 1.3, the Presentation Manager provides a basis for the implementation of object-oriented software engineering principles, allowing the developer to take advantage of the benefits inherent in the object-oriented approach. However, the Presentation Manager application model lacks a built-in inheritance hierarchy, and therefore prevents the developer from realizing the productivity and consistency benefits that may be achieved under the principle of inheritance.

OS/2 Version 2.0 extends the object-based Presentation Manager user interface with the introduction of the **Workplace Shell***, and also provides an object-oriented application model that allows applications to exploit the principle of inheritance. The Workplace Shell application model views an application as a series of objects, typically represented by icons on the Workplace Shell desktop, which are manipulated by the user to achieve the required result.

Objects may represent entities such as files, programs or devices, or may be containers that allow the user to logically group related objects. The properties or contents of an object may be examined using a **view** of the object, which is typically implemented as a Presentation Manager window.

The Workplace Shell application model is based upon the **system object model**, which defines a set of classes to form a basic inheritance hierarchy, and a set of protocols for interaction between application objects. The Workplace Shell defines its own object classes that extend the inheritance hierarchy, and an application developer can continue to extend the hierarchy, subclassing existing object classes to create new classes.

The Workplace Shell therefore brings both the end user and the application developer closer to the concept of direct object manipulation, and allows exploitation of the class-based theory of object-oriented programming. The Workplace Shell application model, along with the creation and manipulation of Workplace Shell objects, is described in detail in Chapter 7, "Workplace Shell and the System Object Model."

1.5 Summary

Presentation Manager facilitates the implementation of an event-driven, object-action user interface and provides predefined constructs that enable a consistent, intuitive user interface for multiple applications, in line with the objectives of the Systems Application Architecture Common User Access component. However, in order to gain the fullest benefit from such an interface, the application developer must adopt a certain degree of object-oriented principles in the design and implementation of applications.

In order to support the implementation of an event-driven interface and facilitate the incorporation of object-oriented design techniques, Presentation Manager provides an object-based, event-driven execution environment with an application architecture that conforms closely to object-oriented theory, within the framework of the Systems Application Architecture Common Programming Interface. Windows become the handles by which the application references data objects, and windows communicate with one another and with the user in an event-driven manner. With the addition of the Workplace Shell in OS/2 Version 2.0, the user and the programmer may deal directly with objects and take full advantage of the concept of inheritance.

Benefits to be gained from the adoption of such principles include enhanced opportunity for code reuse with consequent reductions in development costs, and easier containment of change through encapsulation and data isolation. As the programmable workstation becomes more widely utilized as the platform for line-of-business applications, the importance of sound software engineering principles in the design and implementation of workstation applications will increase, in accordance with the requirement to be able to adequately manage and maintain these applications. OS/2 and Presentation Manager together with the Workplace Shell, which extends the paradigm to further exploit object-oriented concepts, provide a platform for the implementation of such principles.

It must be emphasized that Presentation Manager provides an application architecture at the operating system level which supports the implementation of certain object-oriented software engineering principles, and provides many of the facilities required by such an approach. However, while Presentation Manager supports an object-oriented approach to application design, it does not *force* the application developer to conform to object-oriented design practices. Presentation Manager does not provide, nor does it seek to provide, a complete development environment for object-oriented applications; the provision of such function is the responsibility of application-enabling products that may reside and execute in the Presentation Manager environment.

The remainder of this document will further explore the relationship between OS/2 Version 2.0, Presentation Manager and object-oriented programming, and examine the techniques by which object-oriented applications may be

implemented in the Presentation Manager environment, using both the Presentation Manager and Workplace Shell application models.

Chapter 2. Operating System/2

This chapter briefly explains the differences between the PC DOS and Operating System/2 (hereafter referred to as OS/2) environments, and describes the features and capabilities of IBM OS/2 Version 2.0. The chapter discusses OS/2's retention of compatibility with existing DOS applications, while providing support for multiprogramming and multitasking, larger memory, multiple concurrent communications, etc.

2.1 History

IBM and Microsoft** introduced OS/2 in 1987 as a successor to the PC DOS/MS DOS** operating system[2] in the programmable workstation environment. In the years since its inception in the early 1980s, DOS has grown in both capabilities and sophistication, but by 1987 advanced workstation users were demanding more sophistication from their applications, to an extent which was beyond the capabilities of DOS to deliver.

The choice for operating system developers lay between further enhancing the existing DOS architecture to support more powerful processors, larger memory and so on, or migrating to a new, more powerful operating system architecture which offered more facilities to satisfy user requirements, a broader platform for application development, and potential for future expansion. The latter choice was taken, and the result was OS/2.

The OS/2 operating system environment provides a great deal more power and flexibility than the DOS environment, while maintaining a level of compatibility with existing DOS applications and data. Enhancements made in OS/2 Version 1.3 include:

- Effective use of the advanced capabilities of the Intel 80286 processor
- Support for system memory above 640 kilobytes (KB)
- Support for multiprogramming and multitasking
- Dynamic linking for system and application modules.

In addition, numerous other functions are provided to support and complement these capabilities.

OS/2 Version 2.0 was developed as an extension of the original 16-bit implementation used in OS/2 Version 1.3, and is an advanced 32-bit multitasking operating system for machines equipped with the Intel 80386** or compatible processors. The following new features are implemented in OS/2 Version 2.0:

- Support for the Intel 80386 32-bit microprocessor instruction set; previous versions of OS/2 only supported the 80386 in 80286 emulation mode.
- 32-bit memory management with a flat memory model; previous versions of OS/2 required applications to use the segmented memory model. See 2.3, "Memory Management" on page 12 for further information.
- Enhanced hardware exploitation.

2 For simplicity, the term "DOS" will be used throughout this document to refer to both the PC DOS and MS DOS products.

- Support for multiple concurrent DOS applications with pre-emptive multitasking and full memory protection.
- Support for Microsoft Windows** applications.
- New 32-bit programming environment.
- Binary-level compatibility with previous versions of OS/2, allowing 16-bit applications written for previous versions to execute under Version 2.0 without modification.
- An enhanced Presentation Manager user shell, known as the Workplace Shell, which implements the 1991 IBM Systems Application Architecture CUA Workplace Environment.

The remainder of this chapter describes the features of OS/2 Version 2.0, and also makes reference to architectural features implemented in previous versions of OS/2 where appropriate.

2.2 Intel 80386 32-Bit Microprocessor Support

The basis for OS/2 Version 2.0 is its support for the Intel 80386 microprocessor; previous versions of OS/2 were developed for the Intel 80286 processor, and supported the 80386 in 80286 emulation mode only. Full support of the 80386 means that a powerful set of 32-bit features now becomes available to the operating system and applications, including enhanced memory management and more sophisticated multitasking capabilities. The Intel 80386 and 80486 offer significant improvements over the previous generation of 16-bit microprocessors, while retaining compatibility with these processors.

The memory addressing capacity of the 80386 processor is significantly greater than that of the 80286:

- 4 gigabyte (GB) physical address space; this compares with the 640 kilobyte (KB) address space of DOS and the 16 megabyte (MB) address space of OS/2 Version 1.3.
- 64 terabyte (TB) virtual address space; DOS does not support virtual memory, and OS/2 Version 1.3 supports 2 GB of virtual memory.
- 1 byte to 4 gigabyte memory objects; this compares with a 64 KB maximum size under DOS or OS/2 Version 1.3.

OS/2 Version 2.0 uses many of these processor features and capabilities to provide a more powerful and flexible operating system platform. Note that OS/2 Version 2.0 does not implement the full 64 TB virtual address space provided by the 80386, since this requires use of the segmented memory model; OS/2 Version 2.0 uses a flat memory model, as described in 2.3, "Memory Management."

2.3 Memory Management

Memory management is the way in which the operating system allows applications to access the system's memory. This includes the way in which memory is allocated, either to a single application or to be shared by multiple applications. The operating system must check the amount of memory available to an application, and must handle the situation where there is insufficient free memory to satisfy an application's requests.

Memory management under DOS and OS/2 Version 1.3 was achieved using units of memory known as **segments**, which could be from 16 bytes to 64 KB in size. The memory model implemented by these operating systems was therefore known as a **segmented memory model**. The use of data structures larger than 64KB required the use of multiple segments, the management of which was the responsibility of the application. This led to increased size and complexity, and reduced performance in applications which handled large data structures.

In OS/2 Version 2.0, memory management has been enhanced to provide a **flat memory model**, which takes advantage of the 32-bit addressing scheme provided by the Intel 80386 architecture. This means that through memory management, the system's memory is seen as one large linear address space of 4 GB. Applications have access to memory by requesting the allocation of **memory objects**. Under OS/2 Version 2.0, these memory objects can be of any size between 1 byte and 512 MB. The use of a flat memory model removes the need for application developers to directly manipulate segments, thereby simplifying application development and removing a significant obstacle in porting applications between OS/2 Version 2.0 and other 32-bit environments such as AIX*.

OS/2 Version 2.0 manages memory internally using **pages**, each of which is 4 KB in size. Each memory object is regarded by the operating system as a set of one or more pages. For practical purposes therefore, memory is allocated in units of 4 KB, although a page may be broken down into smaller parts and may contain multiple memory objects.

One of the useful aspects of paged memory is the way in which memory overcommitment is handled; that is, what happens when there is no more real memory left to load applications or satisfy a request for memory from an application. Under OS/2 Version 2.0, individual pages may be swapped to and from disk storage, rather than entire memory objects. This improves swapping performance, particularly when large memory objects exist in the system. The fixed page size also improves swapping performance since the operating system need not be concerned with moving memory objects about in order to accomodate the various object sizes, as was the case with previous versions of OS/2.

For a more detailed discussion of memory management under OS/2 Version 2.0, readers should refer to *OS/2 Version 2.0 - Volume 1: Control Program.*

2.4 Multiprogramming and Multitasking

A **multiprogramming** operating system allows the concurrent execution of multiple applications in the same machine. A **multitasking** operating system is an extension of the multiprogramming concept, which distributes processor time among multiple applications by giving each application access to the processor for short periods of time. OS/2 implements both multiprogramming and multitasking.

Multitasking may be supported in two forms:

- **Cooperative multitasking** requires the active support of applications running in the system, which must explicitly relinquish control of the processor to allow other applications to execute. This form of multitasking is unreliable

and frequently leads to poor performance, since an ill-behaved application can monopolize the processor.

- **Pre-emptive multitasking** uses a *scheduler* as part of the operating system; the scheduler is responsible for selectively dispatching and suspending multiple concurrent tasks in the system. This form of multitasking is more sophisticated, typically leads to greater overall system throughput. and allows implementation of priority dispatching schemes for various tasks.

Numerous mechanisms exist for providing multiprogramming support under DOS; these include products such as Microsoft Windows. However. since such facilities are ultimately dependent upon the single-tasking architecture of the DOS operating system, they typically provide only limited multitasking capabilities; where pre-emptive multitasking is supported, schedulers are typically primitive and performance is relatively poor. Pre-emptive multitasking is not possible during input/output operations, since these operations are performed by the single-tasking DOS operating system.

OS/2 provides pre-emptive multitasking under the control of the operating system, which is designed to use the multitasking protected mode of the Intel 80286 and 80386 processors. OS/2 implements a pre-emptive task scheduler with a multi-level priority scheme, which provides dynamic variation of priority and round-robin dispatching within each priority level. The dynamic variation of priority is achieved on the basis of current activity, and is intended to improve overall system performance and ensure that the system as a whole responds adequately to user interactions. For circumstances where dynamic variation of priority is inappropriate, the dynamic variation may be disabled using a command in the CONFIG.SYS file, and task priority then becomes absolute. In either case, task priority may be set and altered dynamically using a number of operating system functions available to OS/2 application programmers.

The management of tasks executing in the system is further simplified and streamlined under OS/2 Version 2.0. This is due primarily to the fact that support for processes executing in real mode (such as the DOS Compatibility Box in OS/2 Version 1.3) is no longer required, since the execution of DOS applications is supported using virtual DOS machines which run as protected mode processes. See 2.5, "DOS Application Support" on page 18 for further information.

2.4.1 Application Support

OS/2 Version 2.0 supports concurrent execution of the following types of applications:

- DOS applications, in full-screen mode or in windows on the Presentation Manager desktop
- Microsoft Windows applications, in windows on the Presentation Manager desktop
- 16-bit OS/2 applications developed for previous versions of OS/2
- New 32-bit applications developed for OS/2 Version 2.0.

All applications execute as protected mode processes under OS/2 Version 2.0, and are therefore provided with pre-emptive multitasking and full memory protection; each application is isolated from other applications and from the OS/2 Version 2.0 operating system itself.

2.4.2 Processes and Threads

The term *task* (as in multitasking) refers to a hardware-defined task state. While OS/2 supports multitasking, it does not directly use the concept of a task as defined by the Intel 80386 processor architecture. Instead, OS/2 makes a differentiation between **processes** and **threads**.

2.4.2.1 Processes

A process is most easily defined as a program executing in the system. Since it is possible for a single program to be invoked multiple times in a multitasking system such as OS/2, multiple processes may be executing the same program, and each such process is known as an *execution instance* of the program. A process owns system resources such as threads, file handles etc, and a memory map that describes the region of memory owned by that process. Since each process owns its own resources and memory map, which are administered by the operating system on behalf of the process, the resources of one process are protected from access by any other process. In situations where communication between processes is required, OS/2 provides a number of architected mechanisms by which they may be achieved. These mechanisms are described in 2.4.3, "Interprocess Communication and Synchronization" on page 16.

2.4.2.2 Threads

A thread is the unit of dispatching for the operating system's scheduler, and therefore equates closely with the notion of an 80386 task as defined by the processor architecture. Each thread is owned by a process, and a single process may have multiple threads. When a process is created, one thread (known as the *primary thread*) is always created to run the code specified in the process creation system call. Thus a process always has at least one thread. Secondary threads are often used to perform lengthy operations such as document formatting, remote communications etc, thereby allowing the primary thread to continue interaction with the user. *Secondary threads* may be created and terminated at any time during execution of a process. When the primary thread of a process is terminated, the process itself terminates.

A thread executes for a short period of time before the operating system's scheduler preempts the thread and gains control. The scheduler may then determine that there is some other thread that ought to run; if so, the scheduler saves the task state of the current thread and dispatches the new thread, which executes for a period of time until it too is preempted and control returns to the scheduler.

OS/2 Version 2.0 supports up to 4096 threads within the system. Note that this limit includes those threads used by the operating system and by applications executing under operating system control, such as the print spooler. The number of threads available to applications will therefore be somewhat less than 4096.

Since each thread is owned by a process, all threads within that process share the resources and memory map belonging to that process, and thus have access to those resources. OS/2 does not protect memory resources from being accessed by multiple threads within the same process: this is the responsibility of the application developer. However, OS/2 provides a number of architected mechanisms to aid the application developer in maintaining the integrity of the application's resources.

2.4.3 Interprocess Communication and Synchronization

Since OS/2 provides support for concurrent execution of multiple processes, with memory protection between these processes, it must also provide mechanisms to facilitate synchronization and communication between different processes and threads executing in the system, which may wish to share data and control information. OS/2 provides a number of such mechanisms, as follows:

- Shared memory
- Queues
- Pipes (both named and anonymous)
- Presentation Manager messages
- Semaphores.

These mechanisms allow application developers to implement applications using multiple processes or threads, while retaining the ability to communicate data and control information in a controlled manner, and to achieve synchronization between various components of an application.

2.4.3.1 Shared Memory

The OS/2 memory management architecture utilizes the protect mode of the Intel 80386 processor to achieve memory isolation between processes. A process has addressability only to its own memory objects. However, in certain circumstances processes may wish to communicate and pass data to each other; OS/2 allows this by the use of *shared memory objects*. Shared memory objects are dynamically requested from the operating system by the application during execution, and are flagged as shareable by OS/2. It is the responsibility of the applications concerned however, to correctly synchronize the flow of data between processes. OS/2 provides a number of mechanisms by which this synchronization may be achieved. Shared memory and its usage is discussed in the *IBM OS/2 Version 2.0 Application Design Guide*.

2.4.3.2 Queues

Queueing system calls are implemented by a system service routine that uses shared memory and semaphores (see below) for serialization. A queue is created by a process that then becomes the owner of that queue; *only* the owner may read from the queue. Other processes may write to the queue, but only the owner may look at elements on the queue, remove elements from the queue, purge or delete the queue. Queues may be specified with FIFO (first-in, first-out) or LIFO (last-in, first-out) dispatching priority.

A queue has a name, similar to a file name, by which it is known to both processes and by which it is referred to when the queue is first accessed by a particular process. A series of operating system functions is provided by OS/2 to create and access queues. Queues are discussed in detail in the *IBM OS/2 Version 2.0 Application Design Guide*.

2.4.3.3 Pipes

A pipe is a FIFO data structure that permits two processes to communicate using file system I/O calls. The first process writes data into the pipe and the second process reads the data from the pipe. However, the data is never actually written to an external file, but is held in a shared area in memory.

A pipe may be *named*, in which case it has a name similar to a file name which is known by both processes, or it may be *anonymous* in which case read and

write handles to the pipe are returned by the operating system when the pipe is created. It is then the responsibility of the creating process to communicate these handles to other threads or processes.

The creation of pipes is achieved using a number of OS/2 function calls; once created, pipes are then accessed using file system I/O functions. Pipes and their manipulation are discussed in the *IBM OS/2 Version 2.0 Application Design Guide.*

2.4.3.4 Presentation Manager Messages

In the OS/2 Presentation Manager programming environment, application routines known as window procedures communicate by receiving messages from one another and from Presentation Manager itself. Messages may be passed between window procedures executing in the same thread, between different threads in a process, or between processes.

Messages may be used to pass data between routines executing in different threads or processes, or to communicate events in order to achieve synchronization between threads and/or processes. Presentation Manager messages may be used to invoke processing routines in either a synchronous or asynchronous manner. The Presentation Manager messaging model conforms closely to object-oriented programming practices, and is described further in Chapter 4, "The Presentation Manager Application Model."

2.4.3.5 Atoms

Where character strings must be passed between threads, it is relatively simple to pass a pointer to the character string, since all threads within a process share access to memory objects. Where strings must be passed between processes however, more complex methods such as shared memory must normally be used. OS/2 provides a way to simplify the passing of strings between processes, using **atoms**.

An atom is effectively a "handle" to a string that is stored in an area of shared memory known as an **atom table**. Atom tables are maintained by the operating system, and may be private to a particular process or shared by all processes in the system. OS/2 creates a system atom table at system initialization time, which is accessible by all processes in the system.

A process may add a string to an atom table, and obtain an atom that may subsequently be used to access the string. Atoms that reference strings in the system atom table may be passed between processes using any of the methods described in the foregoing sections, and used by another process to obtain the contents of the string.

2.4.3.6 Semaphores

OS/2 applications may be implemented using multiple threads within one or more processes. Within a single process, the OS/2 memory management architecture provides no memory protection for different threads, and hence multiple threads may have addressability to the same data areas. It is important that the integrity of resources such as common data areas or files, shared between threads, be protected at all times. Such resources must be accessed in a serialized fashion. Although OS/2 provides no automatic protection for data resources between threads within a process, OS/2 allows an application to achieve this serialization of access by using semaphores.

A semaphore is a data structure that may be "owned" by only one thread at any time. Semaphores may be used as flags by an application, to indicate that a data resource is being accessed. A thread may request ownership of the semaphore; if the semaphore is already owned by another thread, the requesting thread is blocked until the first thread releases it.

OS/2 Version 2.0 provides a number of different types of semaphores, to be used in different circumstances:

- **Mutex** semaphores provide mutually exclusive access to a particular resource such as a shared memory object. These semaphores offer a useful means of synchronizing access to such resources between different threads or processes.
- **Event** semaphores are used to signal system or application events. These semaphores provide a means of signalling events to other threads or processes, allowing such threads to suspend their execution and "wait" for a particular event to occur.
- **MuxWait** semaphores may be used when waiting for multiple events to occur or multiple mutex semaphores to clear.

Within these semaphore types, OS/2 Version 2.0 provides both **private** and **shared** semaphores. The *system semaphores* and *RAM semaphores* provided by previous versions of OS/2 are also supported, retaining compatibility with applications developed for previous versions of the operating system. Each process in the system may have up to 65535 private semaphores, and there may be up to 65535 shared semaphores in the system.

OS/2 Version 2.0 provides a number of operating system functions allowing the creation and manipulation of semaphores. Semaphores are discussed in the *IBM OS/2 Version 2.0 Application Design Guide*.

2.5 DOS Application Support

OS/2 Version 1.3 provides the capability for a single DOS application to be executed in the system using a facility known as the **DOS Compatibility Box**. The DOS application executes in real mode, and is automatically suspended if the DOS Compatibility Box is switched to the background; that is, pre-emptive multitasking is not supported in the DOS Compatibility Box under OS/2 Version 1.3.

OS/2 Version 2.0 provides the capability to pre-emptively multitask DOS applications along with OS/2 applications, using the **Multiple Virtual DOS Machines** feature of OS/2 Version 2.0. The DOS support has been totally rewritten in OS/2 Version 2.0 and allows multiple concurrent DOS applications where each is executed as a single-threaded, protected mode OS/2 program. This method of implementation provides pre-emptive multitasking for DOS applications, and allows normal OS/2 levels of memory protection; that is, it provides isolation of system memory and other applications, protection from illegal memory accesses by ill-behaved applications, and the ability to terminate sessions where applications are "hung."

DOS support is achieved through the use of virtualization techniques, allowing the creation of multiple instances of separate, independent virtual DOS machines. Through this technique, a virtual interface is provided to each DOS

machine, giving the impression that each application owns all of the required resources, both hardware and software.

Each virtual DOS machine has more memory than the DOS Compatability Box implemented in previous versions of OS/2, and OS/2 Version 2.0 supports the use of Lotus**-Intel-Microsoft (LIM) expanded memory (EMS) and extended memory (XMS) to provide additional memory for those DOS applications that are capable of using such extensions. OS/2 Version 2.0 maps this extended or expanded memory into the system's normal linear memory address space, and manages it in the same manner as any other allocated memory.

The ability of a virtual DOS machine to run within a Presentation Manager window provides immediate productivity gains to existing DOS applications, since they may utilize Presentation Manager desktop features. These features include window manipulation and the ability to cut/copy/paste information between applications using the clipboard.

Application compatibility in the virtual DOS machine is also enhanced over previous versions of OS/2. The virtual DOS machine can be used to execute DOS-based communications applications and other applications that address hardware I/O devices, through the use of virtual device drivers that map the device driver calls from the applications to the appropriate physical device driver within the operating system. Applications using hardware devices that are not required to be shared with other applications in the same system may be accessed using the standard DOS device drivers, without the need for a virtual device driver. Certain restrictions still apply with respect to communications line speed and time-critical interrupt handling.

For applications that require specific versions of DOS in order to operate, OS/2 Version 2.0 provides the capability to load a physical copy of that version into a virtual DOS machine. This provides compatability for those applications that internally manipulate DOS data structures or that use undocumented interfaces.

Application compatability in a virtual DOS machine is further enhanced by the **DOS settings** feature, which allows virtual DOS machines to be customized to suit the requirements of the applications running in them. Properties such as video characteristics, hardware environment emulation, and the use of memory extenders can all be customized using this feature.

Multiple Virtual DOS Machines is described in more detail in *OS/2 Version 2.0 - Volume 2: DOS and Windows Environment*.

2.6 Microsoft Windows Application Support

OS/2 Version 2.0 provides the capability for Microsoft Windows applications to run under OS/2 Version 2.0, in virtual DOS machines. This support allows applications written for Windows 3.0 and previous versions of Windows to coexist and execute concurrently in the same machine.

Each Windows applications executes as a protected mode process. Windows applications are therefore subject to the full memory protection facilities provided to protected mode applications under OS/2 Version 2.0, and are protected from one another and from DOS or OS/2 applications executing in the system. This is in contrast to the native Windows 3.0 environment, where limited

protection is provided for Windows 3.0 applications, and none at all for DOS applications unless Windows is running in enhanced mode.

The execution of Windows applications as protected mode tasks also allows these applications to take full advantage of the pre-emptive multitasking capabilities of OS/2 Version 2.0, with full pre-emptive multitasking between Windows applications, DOS applications and OS/2 applications. This is again in contrast to the native Windows 3.0 environment, where pre-emptive multitasking is available only for DOS applications, only when Windows 3.0 is running in enhanced mode, and only when no input/output operations are being performed, thereby impacting performance and preventing many applications written for previous versions of Windows from executing. OS/2 Version 2.0 has no such restriction.

As with DOS applications, Windows applications may make use of EMS and XMS memory extenders in order to access memory above 640 KB. This support is provided in an identical manner to that provided for DOS applications.

Support for Microsoft Windows applications under OS/2 Version 2.0 is discussed in more detail in *OS/2 Version 2.0 - Volume 2: DOS and Windows Environment*.

2.7 Dynamic Linking

OS/2 system services are requested by application programs using function calls; the external code references generated by such calls are resolved when the program is loaded or when the segments of the program are loaded, rather than at link-edit time. This deferred resolution of external references is known as **dynamic linking**, and is available to applications, which may incorporate their own routines into **dynamic link libraries (DLLs)**.

Dynamic linking may be achieved in two ways under OS/2;

- **Load-time dynamic linking** resolves external references within a code segment at the time the segment is loaded into memory.
- **Run-time dynamic linking** postpones the resolution until the actual execution of the code, at which time the appropriate module is explicitly loaded and invoked by the application.

Load-time dynamic linking is the simplest mechanism; as already mentioned, OS/2 system services are implemented in this way. Load-time dynamic linking is used whenever an application developer wishes to provide common services that may be used by multiple applications, and which are implemented independently of the applications that will use them. Run-time dynamic linking is used where particular routines may or may not be used by an application, and thus should not be loaded into memory unless required. If an application requires that such a routine be executed, the application may then explicitly load the routine and execute it.

Dynamic linking provides an architected method of extending the services of the operating system; all of the application programming interfaces supported by OS/2 are implemented using dynamically linked modules. An application developer may use the same facilities to create his or her own dynamically linked modules to provide additional services or standard routines that may be used by applications executing in the system.

Dynamic linking is of benefit in that routines contained in DLLs are completely independent of application code, and may be modified without the need to re-link applications. In this way, DLLs contribute to the containment of change within applications. In addition, the contents of DLLs are not limited to application code. Presentation Manager **resources** such as icons, bitmaps, graphics fonts, window definitions etc, may be generated and stored in a DLL for subsequent use by applications. See Chapter 9, "Presentation Manager Resources" for a further discussion of Presentation Manager resources. DLLs thus provide a powerful mechanism for the creation of reusable modules for both full-screen and Presentation Manager applications.

Secondly, the creation of DLLs as re-entrant routines reduces the storage requirements for OS/2 applications, since multiple applications may make use of the same memory-resident copy of a DLL. This re-entrancy and reusability, in conjunction with the code independence gained by using a DLL, makes the DLL a useful vehicle for implementation of standard service routines, which may be accessed by any application or process within the system. This contributes to standardization within the business organization, which in turn can result in improved productivity and reduced maintenance effort since the code to implement a particular function need reside in only one location.

Note that a DLL is *not* a process under OS/2. The re-entrant nature of a DLL allows multiple applications to use the same memory-resident copy of the code; however, each instance executes under the control of the process that invoked it.

2.8 Summary

OS/2 provides the programmable workstation platform for the delivery of Systems Application Architecture application functionality in the standalone workstation and cooperative processing environments. OS/2 overcomes the limitations of the DOS operating system by providing support for large physical and virtual address spaces and supporting concurrent execution of multiple applications with memory isolation and automated task dispatching. While enforcing memory protection between applications, OS/2 provides architected mechanisms to allow interprocess communication and data sharing in a controlled manner.

OS/2 also provides compatibility with existing DOS applications, since OS/2 Version 1.3 allows a single DOS application to run using the DOS Compatibility Box, and OS/2 Version 2.0 allows multiple concurrent DOS applications to execute in virtual DOS machines. OS/2 Version 2.0 also supports Microsoft Windows applications in a similar manner to DOS applications. This allows users of OS/2 systems to continue to use their existing DOS and Windows applications under the new operating system.

OS/2 also implements dynamic linking, which allows an application developer to isolate common application services in separate modules known as dynamic link libraries (DLLs). Calls to application services provided in a DLL are resolved at execution time, which means that any modifications to the routines contained in a DLL do not require any maintenance to applications using that DLL. In addition, DLLs are created as re-entrant code, thus allowing multiple applications to use the same memory-resident copy of the DLL code and thereby reducing storage requirements.

OS/2 provides an operating system environment for the programmable workstation that enables a far greater degree of functionality and sophistication on the part of application programs. OS/2 Version 2.0 provides architected methods for overcoming most of the inherent limitations of the DOS and OS/2 Version 1.3 environments, and providing the workstation user with a higher level of capability in the workstation. OS/2 provides the vehicle that will enable the fulfillment of the Systems Application Architecture cooperative processing direction.

Chapter 3. Object-Oriented Applications

This chapter provides a brief overview of some concepts involved in object-oriented application design and programming, and the way in which this approach differs from traditional top-down functional decomposition. It is not intended as an indepth analysis of object-oriented programming, since such a task is beyond the scope of this document, but serves merely to provide a background against which the implementation of object-oriented principles under Presentation Manager may be discussed.

3.1 Object-Oriented Concepts

Object-oriented application design places the focus of an application on the logical entities or *objects* (typically items of data) upon which a program will operate, and attaches procedures or routines to manipulate these objects. A logical data **entity** (such as a group of records) may have multiple representations within the system. Each of these representations is known as a **data object**, and each data object may have a finite set of **actions** performed upon it. The data object itself, along with the routines (known as **methods**) used to perform these actions, are together regarded as an **application object**.

Note that in the remainder of this document, the term **data object** will be used to denote a particular representation (for example, on the screen or in a disk file) of a logical data entity. The term **application object** will be used to denote the conjunction of a data object and its methods. While these terms are not in general use, they will be used here in order to provide a distinction between a data item, and the conjunction of that data item and the routines that act upon it.

Application objects typically respond to **events**, which originate outside the object and which may be system-initiated or user-initiated. The sequence of these events determines the sequence of operations within the application, and the progression of dialog between the application and the user, rather than the application determining the sequence of the dialog, as is traditionally the case. Such an object-oriented application environment is thus said to be **event-driven**.

Events are communicated to an application object by means of a series of defined **messages**, which are *not* considered to be part of the objects between which they are passed. The focus of the program thus becomes the object, rather than the procedure, and the program becomes a flow of messages between cooperating objects.

Actions on a data object should be possible *only* by sending messages to the associated application object; an object's methods should not be directly accessible from another object, nor should the behavior of an object be dependent upon any external source other than the messages it receives. A message should also specify only the action that is to be carried out, and not the way in which it is to be accomplished. It is the responsibility of the receiving object to determine the way in which to carry out a requested action. Consequently, the behavior of an application object may differ, depending upon the class and content of its input messages. A corollary of this statement is that the result of passing the same message class may vary, depending on the target object class and its interpretation of that message. These guidelines outline the concept of **polymorphism**.

One of the primary properties of an application object is its modularity; objects that obey the foregoing rules should be largely independent of one another, and the implementation of one object should not be dependent upon the internal details of another. Data belonging to an application object should be accessible only by that object; requests for access by other objects should be made via appropriate messages sent to the application object that "owns" the data object. Thus the only information necessary to use an application object is a knowledge of the messages it can receive and operate upon (through its methods). This rule encompasses the principle of **encapsulation**, which states that a data object should be defined, created and/or accessed solely from within its "owner" application object.

Since a number of application objects may exist with similar characteristics, such objects are usually grouped into **object classes**. A class consists of those objects that share similar properties and methods. An object class is typically associated with a single data object or type of data object, and has a defined, finite set of methods associated with it. It is the class that normally defines the messages and methods applicable to an object. Each object belonging to a particular class is then known as an **instance** of that class. Each instance inherits data objects and values defined for its class, and may also contain its own data, known as **instance data**; the properties of instance data are typically obtained from the definition of the object class, but the values are defined uniquely by each instance.

The object-oriented approach is most suited to situations where the purpose of the application is the manipulation of data, whether on the screen or in a data file, and where the exact sequence of actions is not critical, provided all necessary actions are carried out. The focus of an object-oriented application is the data objects that are being manipulated; the functions to be performed are subordinate to the data objects upon which they will act. In addition, the event-driven user interface places the user in control of the sequence of actions, and provides flexibility with respect to the way in which the desired result is achieved.

An advantage of the object-oriented design approach for data manipulation applications is that a particular data object is "owned" by one application object, and that the definition of and establishment of access to that data object is achieved from within the application object. Since application objects may be made relatively independent of one another, other objects may be isolated from changes to the data structure. This greatly simplifies application maintenance, since all necessary changes need only be made within a single object.

Note that since application objects are closely related only to their associated data objects, and not to the applications from which they are accessed, it follows that application objects may be constructed for each data object, and accessed from multiple applications. An application need not be aware of the internal workings of an application object, but need only know the correct type and format of the messages through which to interact with the object. Thus the object-oriented approach facilitates code reusability.

3.1.1 Object-Oriented vs Functional Decomposition

Under a traditional **functional decomposition** approach to program design, often referred to as **structured programming**, the function or procedure is the unit of modularity; programs are designed and implemented by placing a number of well-defined procedures in a particular order, and executing these procedures to achieve a desired result. The focus of the design is the procedure or action to be performed. Objects such as data structures are attached to procedures and passed between them using parameters. A user typically selects an action to be performed, and then selects or enters a data object upon which to perform that action.

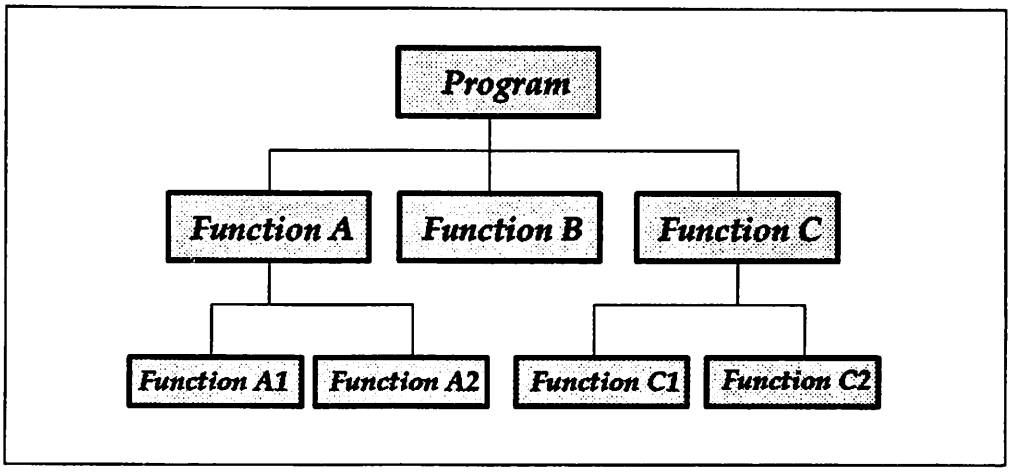

Figure 1. Program Flow - Functional Decomposition Approach

The functional decomposition approach is best suited to situations where the procedure is necessarily the focus of the application (for instance, a process management application), and where the correct sequencing of operations to be performed is a crucial factor in the successful execution of the required task. Under this approach, the application defines the sequence of actions which the user performs; that is, the application controls the user interface.

In an object-oriented approach, the application object is the unit of modularity. Application objects communicate with each other and pass messages containing actions to be performed. Object-oriented programming is hence the conceptual inverse of functional decomposition, and is a logical extension of the industry trend toward data-centric application design.

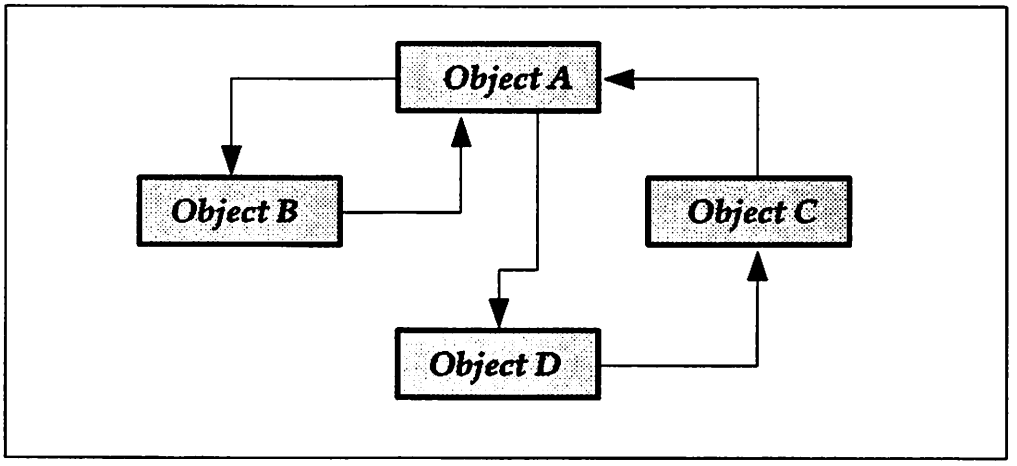

Figure 2. Program Flow - Object-Oriented Approach

This is not to say that an object-oriented application should not be structured. Although such an application consists of objects that are largely independent of one another in programming terms, normal structured coding techniques should be followed in the creation of the methods within each application object.

The object-oriented approach also requires firm management of the application development process in order to achieve the greatest possible level of productivity through code reuse. Administration and control of existing objects is vital in order to allow application developers to access and use these objects in their applications. Management of the application development process is discussed in greater detail in Chapter 18, "Managing Development," and the structuring of application source modules in order to provide optimal granularity is described in Appendix B, "Application Program Construction."

While it is possible for an application designed according to functional decomposition principles to implement some of the characteristics of object-oriented applications such as message passing, such applications should not be regarded as truly object-oriented. If the design approach centers on procedures rather than data objects, then the application is designed along functional decomposition guidelines. In this case, message passing is merely a replacement of the normal subroutine call mechanism, and does not significantly affect the structure of the application.

3.1.2 Class-Based vs Module-Based

The notion of object classes, and the extent to which this concept is taken, provides the distinguishing factor between two primary schools of thought within the object-oriented paradigm. Under a **class-based** approach, objects are defined in terms of their class, and each class is defined in terms of other previously defined classes, the properties and methods of which are automatically conveyed upon the new class; this is known as the principle of **inheritance**.

For example, the object class "horse" may be defined as a sub-class of the object class "quadruped," with the additional properties of being able to be ridden and eating grass. A further object class "pony" may then be defined as being a sub-class of the class "horse," with an additional upper limit on size. While this is a somewhat frivolous example, it illustrates the principle that an object class is defined in terms of other object classes, and need only explicitly define those properties and methods that are unique to that object class. All other properties and methods are inherited from its parent class or classes. This introduces the concept of an **inheritance hierarchy**, in that an object inherits not only the properties and methods of its class, but also those of other classes by way of which that class was defined.

The major advantage of such an inheritance hierarchy is that, given a well-documented set of existing objects, it becomes extremely easy to create new object classes, simply by defining the new class in terms of other classes that already exist, and simply specifying any new or different properties or methods that apply to the new class. This of course assumes the use of adequate object documentation and management practices. Without such practices, it becomes difficult if not impossible to identify a suitable base object from a large library of existing object classes.

However, many existing implementations of the class-based approach extend the inheritance hierarchy to a great degree, such that almost all imaginable object

classes are defined in terms of parent object classes. While this provides a unified approach to the problem of object definition, the significant disadvantage of such an approach is the increased level of interdependence between objects. The unit of modularity becomes the complete hierarchy rather than the individual object, since an object has no complete definition in its own right. The reuse of a single object therefore requires the inclusion of its complete parent hierarchy. Since it is typical for this parent hierarchy to be defined dynamically using run-time definitions for parent classes rather than defined statically at application generation, it is also possible for changes to a parent class to cause unforeseen side-effects in the behavior of descendant object classes. Thus inheritance hierarchies require careful management to ensure that such side effects do not occur and adversely affect the integrity of applications and data.

Where the inheritance hierarchy is taken to the extent of providing system-defined object classes, to which all application-defined object classes are linked, the hierarchy and thus the application is dependent upon the existence of a *virtual machine* conceptual environment, which must also be accepted along with the hierarchy. This in turn may result in significant penalties in terms of application efficiency and run-time performance.

A **module-based** approach to object-oriented programming defines each object as complete in its own right. Objects may still be grouped into classes for easier definition and processing, but each class possesses its own complete set of properties and methods, and is not dependent upon another class for a part of this definition. The primary advantage of the module-based approach is the increased level of independence between objects, with a finer degree of granularity in the application allowing object reuse with a lower level of overhead. The main disadvantage of this approach is that each object class must be completely defined, requiring more work on the part of the application developer at the time the object is created.

The concept of inheritance, while providing great potential for productivity enhancement during the application development process, must be carefully managed in order to avoid additional complications in application management and maintenance due to object interdependencies. Side effects arising from modification to parent object classes may adversely affect the integrity of an application. The alternative course of action, that of prohibiting the modification of existing objects in favor of creating new objects that inherit only the unmodified properties of the existing object, is often not viable due to the increased application overhead and managerial effort required to maintain and control an ever-expanding inheritance hierarchy. Reliance on the behavior of existing objects must therefore be viewed with extreme caution in the absence of effective management controls over object modification.

The increase in development productivity provided by the use of inheritance may often be offset by the increased time and effort spent in regression testing of existing applications in order to determine any effects on these applications caused by the modification of existing object classes. Tight managerial controls over development must therefore be maintained in order to identify and isolate those existing object classes that are modified and which are likely to affect existing applications.

One technique that can be used to minimize the impact on existing applications is for the development organization to adopt a standard whereby, once an application object is deployed in a production environment, new applications that

use the object may *only* modify its behavior through the use of subclassing (see 3.1.3, "Subclassing" on page 28). This means that the object itself is not modified, and other applications that use the object are not affected.

3.1.3 Subclassing

As already mentioned, the object-oriented approach facilitates the reuse of application code; generic objects may be created and used by multiple applications to perform actions upon the same data object or type of object. However, one of the criticisms often levelled at this capability is that it becomes impossible to foretell the total set of actions that may ever be required with respect to a particular data object, and that an object is therefore never truly reusable.

The object-oriented approach overcomes this difficulty by providing a way for applications to modify the behavior of existing application objects without modifying the objects themselves; this is known as **subclassing** the object. A subclass application object is composed of a data object definition and a certain number of methods to manipulate that object. The subclass application object may contain methods that are not contained within the original application object created for that data object, or methods that are modified in some way from those contained in the original object. In this way a subclass application object may add new methods to perform actions which are not performed by the original object class, or to handle certain types of message in a different way than that normally carried out by the original object class.

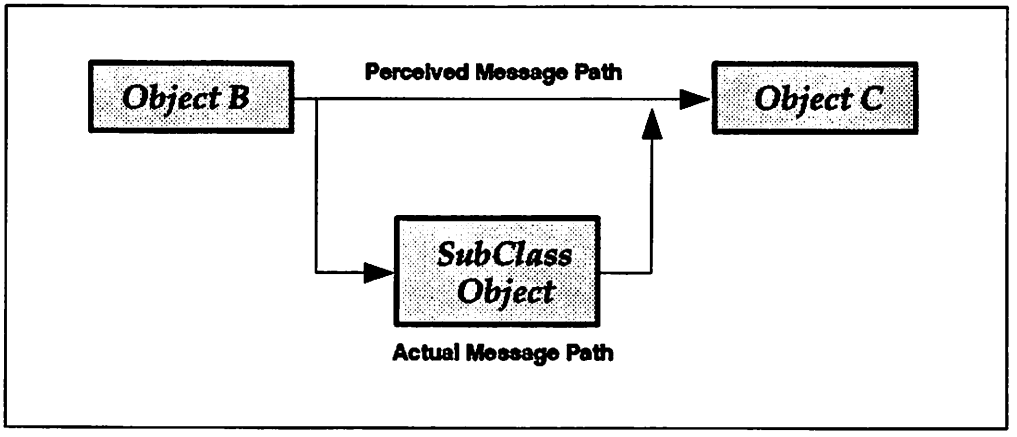

Figure 3. Subclassing an Application Object

When an application object has been subclassed, all messages intended for that object are directed to the subclass application object first. The sending object need not be aware that the message has been diverted. If the subclass application object does not contain a method to deal with a particular message, it should then pass the message on to the original application object, for processing by one of that object's methods. The original application object receiving a message in this way should also be unaware that it has been subclassed.

A useful application of the subclassing principle occurs when a generic application object is defined and stored away for use by many applications, taking advantage of the reusability aspects of the object-oriented approach. Where one application wishes to perform a particular action in a slightly different manner than that performed by the methods associated with the generic object, a subclass application object may be created containing a new method for that

specific action only, and passing all other messages to the original application object for regular processing. It can be seen that subclassing is a technique for application of the concept of inheritance, through its ability to transparently add properties and methods to existing objects.

Subclassing also provides a way to overcome the danger of inadvertently impacting the behavior of other applications by modifying an existing application object. If a standard is adopted whereby existing application objects in a production environment may *only* be modified through subclassing, such changes do not impact applications using the original object or which may themselves have subclassed that object. In this way the need for regression testing of affected applications is eliminated, and the degree of object-management required is significantly reduced.

When a functional requirement may be satisfied by modifying the methods of an existing application object (through subclassing), a decision must be made regarding the relative merits of modifying the object, against creating a new object. Various texts advocate a rule whereby a new object should be created when more than 10% or 20% of an existing object's methods must be modified. However, the decision of whether to modify an existing object or create a new object must be taken on the basis of object complexity, degree of modification and experience.

3.2 User View vs Application View

A primary benefit of the object-oriented approach is its intuitive user interface; a user selects an object and performs a series of predefined actions upon that object. This object-action style of interface encourages the user to explore the application through context-sensitive actions, and reduces the overall complexity of the user's interaction by reducing the levels of hierarchy required for the application.

However, the end user may have a different view of an object-oriented application from that which must necessarily be taken by the application developer. For instance, in the case of a text editor application editing a file, there may in fact be two versions of this file; one existing in memory, being manipulated by the application, and the other stored on a disk. The application would consider these as two separate data objects, each with its own set of methods, and would create two application objects.

Normal user interaction would take place with one object (the memory representation of the file), and when the user selects a "Save" action for the file, a message is passed to the second object (the disk representation of the file) with the information necessary to save updates on disk storage. The user considers the logical data entity as the object being manipulated, while the application must distinguish between the representations of that data entity in various locations within the system. The user's *metaphorical* view of the data object is therefore not carried over to the application's view, which must by necessity be more concerned with the reality of that object's manipulation.

Note however, that this distinction between representations should not be apparent to the end user. The end user should perceive a single object (the text file) upon which he or she would perform actions. Thus there is a distinction between the user's view and the application's view of the objects. An understanding of this distinction is important in order to comprehend the

difference between an object-action user interface and an object-oriented application design.

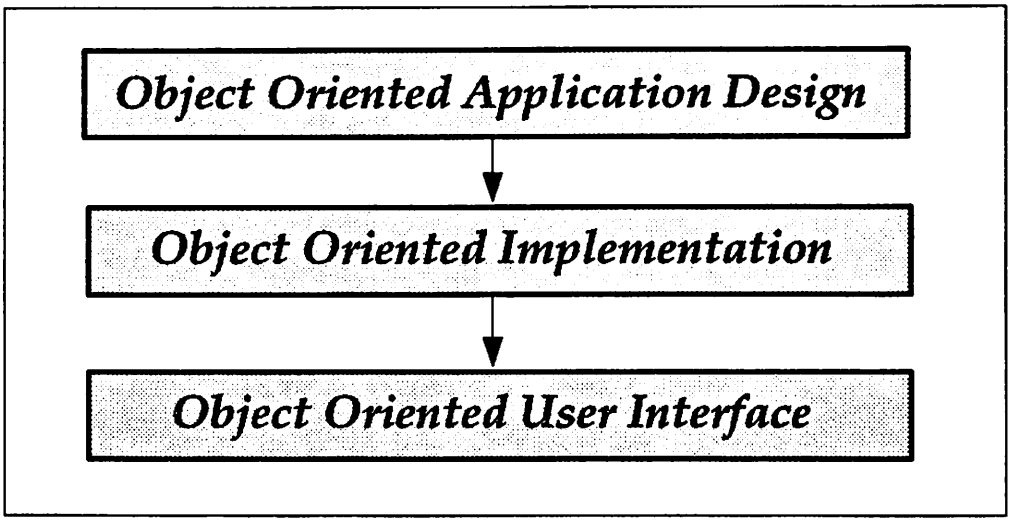

Figure 4. Object-Oriented Development Progression. This diagram shows the interdependence of object-oriented design, implementation and user interface.

The two concepts are complementary but distinct. An event-driven, object-action user interface necessarily emerges from an object-oriented application design and implementation. It is not possible to provide such an interface unless the application structure conforms to a certain level of object-oriented principles and practices. This in turn is dependent upon an object-oriented application design. For this reason, the proper implementation of the graphical user interface concepts defined in the *IBM Systems Application Architecture CUA Advanced Guide to User Interface Design*, requires applications to be designed and implemented using object-oriented guidelines.

3.3 Object-Oriented Design

The success of object-oriented design lies in the correct and intelligent definition of application objects and their methods as coherent and independent units. The secret of a successful approach to this task is the consideration of the data objects themselves as the focus, rather than the procedures that will operate upon these objects. Since the objects are typically associated with data, an entity-relationship model is often a useful starting point.

Correctly-designed application objects facilitate reusability, since the data object and applicable actions are all defined within the application object. Additional applications that require manipulation of that data object may use the existing application object to achieve the required actions. Certain applications may require additional, unforeseen actions, or that existing actions be carried out in a different manner; in such cases, subclassing the application object allows such modifications to be carried out. One of the aims in the high-level design of an object-oriented application should be to make maximum use of existing application objects where possible, in order to reduce the design, coding and testing effort required. This not only reduces the time and expense involved in application development, but enables application solutions to be delivered in a shorter time frame, allowing the business enterprise to respond more quickly to a dynamic marketplace.

The correct definition of application objects and their boundaries also facilitates change management and maintenance of application code, since changes to a particular data object should affect only the application object(s) dealing with that data object. Thus the effects of change are ideally confined to a single application object. Modifications to a method within an application object should not affect the workings of other objects with which that application object interacts, provided the external interfaces of that object are not altered by the modification. This containment of change within a single application object has the potential, in conjunction with proper configuration management techniques, to greatly ease the effort and cost involved in application maintenance.

The following steps are necessary in the design of an object-oriented application:

1. Identify the data objects (that is, the different representations of the logical data entity or entities upon which the application is to operate) and the relationships between data objects.
2. Determine the set of applicable actions to be performed on each data object.
3. Determine whether application objects have been previously been created for the data objects to be manipulated by the application, and to what extent these existing objects perform the required set of actions for each object.
4. Determine the message classes required to achieve the desired communication and to initiate required actions that are not already satisfied through existing application objects.
5. Design the methods necessary to carry out the additional actions.

These steps are generic in nature, and must be combined with suitable management controls, checkpoints and documentation standards to ensure adequate design quality at each stage in the process.

3.3.1 Object Identification

The identification of data objects and their relationships should begin with the definition of a normal entity-relationship model, and the extension of this model to reflect the representations of data objects as well as the logical data entities involved. The objective should be to achieve an optimal balance between the number of object classes and the size and complexity of each class, since each object may itself be composed of other objects, also bearing in mind that multiple objects of the same class may exist. Optimizing the number of object classes will allow the number of message classes to be minimized, which in turn simplifies the design of the methods and the eventual testing of the object classes.

3.3.2 Action Identification

Having defined the data objects and their relationships, the set of actions pertaining to each data object should be determined. Once this point is achieved, the high-level design of the application objects, and therefore of the application itself, is essentially complete, and should be documented and approved by all concerned parties before the detailed design of methods commences. Note that it is not necessary to define *all* of the applicable actions for each data object, since the essentially independent nature of methods allows additional actions to be added to an application object.

3.3.3 Search for Existing Objects

Once the data objects and the set of actions required for each object have been defined, the application designer should determine the existence of any previously-created application object classes for that type of data object. The use of existing application objects, with additional actions and methods handled through subclassing, can significantly reduce the amount of coding required.

The accurate identification of existing application object classes requires that each application object, upon completing its final testing, must be placed in an object library, and its external interfaces (both input and output) must be fully documented and placed in a retrieval system from which the object's description may be recalled by designers of future applications. The organization and level of sophistication of such a system is largely at the discretion of the development organization, but becomes more crucial as the number of application objects grows larger over time. It is strongly recommended that any organization embarking on a strategy of object-oriented application development should adopt an efficient object library management system from the outset.

3.3.4 Message Definition

When the high-level design is complete, the message classes and their contents must be defined for each action that will be performed on and by an object. Note also that the same message class may be used with different object classes to achieve a different result, in accordance with the principle of polymorphism, thus reducing the number of defined message classes and simplifying the design of the application objects and their methods.

The message classes comprise the interfaces between objects, and provide the input and output for the methods associated with the object. These message interfaces must therefore be documented to facilitate reusability of the newly-created application objects by documenting the valid inputs and outputs for each object class, and the behavior of the object in response to these messages.

Since the messages received and dispatched by an object constitute the inputs and outputs required and expected of that object, the documented message interfaces provide a valuable starting point for developing a test plan for the object. Since the inputs, actions and outputs are known, a comprehensive set of test data may then be formulated to test the methods associated with each action, with both valid and invalid message inputs.

3.3.5 Method Design

Traditional functional decomposition techniques may then be used in the design of the methods to complete the required actions. If the high-level design has been completed correctly, the application objects should be relatively independent of one another. It should therefore be possible to complete the development and testing of the methods for each object class as a separate task with each class, its methods and valid interfaces defined and unit-tested before the application is brought together for final integration-testing.

3.4 Object-Oriented Implementations

There has been much discussion in the computing industry press concerning application development products and tools that support the creation of object-oriented applications. While the use of such products is a valuable aid in designing and developing an object-oriented application, it should not be construed tnat their use is essential to the successful implementation of an object-oriented approach.

It is quite possible to implement module-based object-oriented principles to a practical and beneficial degree in the Presentation Manager environment, using a "conventional" programming language such as "C." However, the degree of discipline and amount of work required of the application developer is greater when a standard programming language is used. Object-oriented tools or language extensions make the task of the application developer much easier, since many of the object-oriented concepts are handled by the tools themselves, without the need for the developer to concern him/herself with the details of their implementation.

Object-oriented programming tools fall into two general categories:

- Those that provide extensions to an existing programming language and implement certain object-oriented conventions, such as C++. These languages typically provide object-oriented constructs but do not force an application developer into the use of such constructs. The strength of such implementations is their flexibility, but they have the inherent weakness that it is easy to develop application code that does not conform to object-oriented programming practices.
- Those that provide a complete programming language syntax, which obeys and implements object-oriented principles, such as Smalltalk V**. These languages provide an additional benefit in that they force the application developer into obeying object-oriented programming practices, but at the expense of flexibility.

The tools that are marketed as "object-oriented" and which are currently available in the marketplace tend to implement the class-based approach to object-oriented programming. Their primary benefit is thus that they provide an inheritance hierarchy, and so make the task of object creation much easier for the application developer, albeit at the risk of reduced code efficiency and possible dependence upon a conceptual run-time environment.

The choice of an object-oriented programming tool is very much an individual one, and depends on the requirements and priorities of the development organization, the skills and prior experience of the application developers concerned, and the degree to which object-oriented practices are to be enforced within the organization.

3.5 More Complex Objects

For relatively simple applications where a data object is retrieved, manipulated and saved again, the foregoing definition of an application object will suffice. However, the situation often arises when an application must perform some processing that is rather more complicated. Either the application must interface with another system or external device or a certain work task must be performed in a particular sequence of operations that is critical to the correct completion of

the task. The incorporation of such requirements into the object-oriented paradigm requires an expansion of the application object concept.

The key to successful expansion of the object-oriented paradigm lies in the definition of a data object. Whereas a data object was previously defined as a manifestation of a logical *data* entity, the definition will now be expanded to include any other type of logical entity or source of information, such as an external system or a procedures manual.

3.5.1 Device Manipulation

The entity represented by an application object may be a physical device. For example, suppose an application is required to access a particular type of external data input device such as an MICR reader. The definition of the protocols and specialized access routines necessary to interact with such a device can be encapsulated within an application object, and the various actions performed by the device on behalf of the application (such as getting the next MICR document) then become the methods associated with that object. The application thus regards the MICR reader as an object to which it passes messages in order to perform actions; the results of those actions are then conveyed back to the application, also by way of messages.

3.5.2 Access to Remote Systems

The representation of a physical device as an application object may be extended to encompass any external entity with which an application must interact. Suppose that an application executing in a programmable workstation must interface with a number of server applications executing in a System/370 host, using the SRPI interface. The host system may be regarded as a logical entity, and a single application object created to interact with that entity. All communications-related and interface-specific code may then be encapsulated within that object, and other objects within the application need not concern themselves with the details of the SRPI communication. A message passed to the application object by a requesting object elsewhere in the application will contain the name of the server application to be invoked, and the data to be passed to the server application. A return message from the host-interaction object will contain the reply data from the server application.

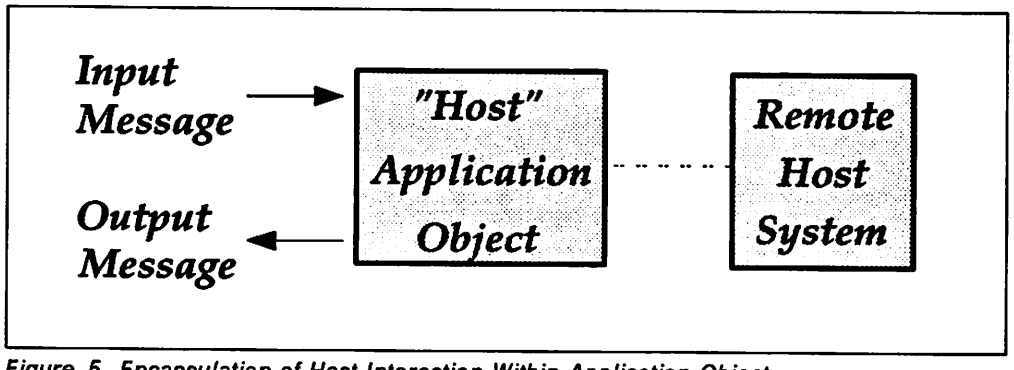

Figure 5. Encapsulation of Host Interaction Within Application Object

The isolation of programming interfaces and protocols specific to the communications architecture, within the "Host" application object provides an easy means of insulating the remainder of the application from changes to communications network configurations or to the remote system itself. Such changes would only require modification to the "Host" object.

3.5.3 Procedure Manuals

An administrative procedure, which is merely a set of information that documents the way in which a task must be performed, may also be regarded as a **procedural entity** that may be encapsulated in an application object. Such an application object typically contains a small number of methods, and may possibly contain only one.

In the case of an object that contains only a single method and merely accepts data from a calling application object rather than possessing data objects of its own, the object may be regarded as simply a method rather than an application object in its own right. Where it is desirable to invoke the procedure from a number of applications, the procedure may be placed in a separate executable module and dynamically bound to its calling applications, thereby maintaining independence of the procedure from the applications.

A message is passed to the application object identifying the action to be performed and providing the necessary input data. The application object will then typically carry out a modal dialog (or possibly a series of dialogs) with the end user to obtain any further information, leading the user through the necessary steps in the required order. When the procedure has been completed, the application object terminates the dialog and possibly passes a message to its caller or to another application object, containing an acknowledgement of completion, or information collected during the procedure. Where completion of a method is mandatory to correct execution of the application, this acknowledgement of completion provides a useful mechanism for the caller to determine that the method has been successfully executed.

If such procedures are correctly defined at a generic level (for example, *Enter the customer data*), their application objects may be stored in a library and used by multiple applications. More complex procedures may be constructed within an application by invoking a number of such application objects in sequence. Acknowledgement messages from the application objects can be used to verify that the required steps have taken place.

Thus it can be seen that the object-oriented paradigm, when the definition of a data object is sufficiently expanded to include all types of logical entity, and when properly applied with correctly designed application objects obeying the aforementioned rules and guidelines, is generally applicable to almost all applications. A wide variety of applications may therefore achieve the modularization and reusability benefits afforded by an object-oriented design approach.

3.6 Summary

It can be seen from the foregoing discussion that the object-oriented paradigm is a logical consequence of the move toward data-centered application design. An object-oriented approach provides many benefits for applications which manipulate data, not the least of which is the ability to implement an intuitive, event-driven user interface where the user manipulates a number of conceptual objects in a metaphorical manner that mirrors the "real-life" manipulation of those objects.

It should be stressed however, that object-oriented programming is not necessarily suited to all applications; its use is recommended only where data is

the central factor in the application's task. There may be situations where the procedure, rather than the data, is necessarily the focus of the application, or where an event-driven style of user interface is not appropriate to the task being performed. In such cases, traditional structured programming techniques hold advantages over object-oriented programming. However, in situations where only a portion of the work task is procedurally oriented, it is possible for the two approaches to coexist within the same application.

Object-oriented programming focuses on the principles of data abstraction and encapsulation, with all access to a data object being controlled by the application object which "owns" that data object. This principle allows the object-oriented approach to facilitate the creation of reusable application objects, since the interface to an object is defined only by the messages it receives and dispatches. The implementation details of the data objects or methods belonging to an application object are typically transparent to other application objects. Thus the effects of changes to these data objects or methods may be contained within a single application object, easing the task of change management and application maintenance.

A distinction must also be drawn between an object-oriented application design and an event-driven, object-action user interface. The two concepts are complimentary in that the provision of a truly event-driven interface for the end user is dependent upon the application being designed and implemented according to object-oriented principles. However, while the two concepts are complimentary, they are not identical and the difference must be borne in mind when designing an application.

Object-oriented application design begins with a definition of logical data entities and their representations (data objects) in the system. Once these data objects are defined, the actions relevant to each may be determined, and the design of the messages to convey each action and the methods necessary to achieve the actions may be undertaken. The definition of messages and actions completes the high-level design of the application, and traditional functional decomposition techniques may then be applied to designing and developing the individual methods. The independent nature of application objects and of the methods within an object facilitates modularization of the application, and allows the design, development and testing of the methods for each object to take place independently of the methods for other objects.

A number of programming languages and tools exist which implement object-oriented programming practices to varying degrees. These may exist in the form of object-oriented extensions to an existing programming language, or as complete programming languages in their own right. The degree to which these products enforce object-oriented practices also varies from one product to the next. It should be stressed however, that the implementation of object-oriented techniques in application design and development is not dependent upon the use of any particular tool or programming language, but rather depends on the correct application of object-oriented design concepts. When such concepts are correctly applied in the design of an application, it is quite possible to develop object-oriented applications in conventional programming languages such as "C." The difference lies in the amount of latitude given to the individual application developer with respect to the interpretation of object-oriented principles.

It may be argued with some justification, that according to many popularly-accepted definitions, the techniques offered in this chapter do not constitute a truly object-oriented application model. However, such concepts as a full inheritance hierarchy are not directly supported by execution environments such as Presentation Manager, and are difficult to provide without the use of additional object-oriented programming tools. It must be stressed that the methodology outlined herein is not intended to be a purist implementation of the full object-oriented paradigm, but is intended to illustrate the application of certain object-oriented principles to the design and implementation of Presentation Manager applications, in accordance with the module-based approach to object-oriented programming.

These principles, when applied to the Presentation Manager environment, afford significant enhancements in the areas of code reusability and application modularity, and subsequent benefits with respect to reduced development time and effort, and easier application maintenance and change management. Additional tools may be utilized to apply further object-oriented principles to Presentation Manager application design, in order to achieve the benefits associated with a class-based approach to object-oriented programming. However, such tools typically have additional drawbacks that must be weighed against their advantages, with regard to the specific development scenario.

Chapter 4. The Presentation Manager Application Model

The Presentation Manager environment lends itself to the implementation of object-oriented programs under the module-based approach to object-oriented design. Presentation Manager provides far more than merely a means of displaying information on the screen. It implements an event-driven, object-based application execution environment and provides many services required for the definition and manipulation of application objects and messages.

This chapter will examine the execution environment provided by Presentation Manager, describe the structure of a Presentation Manager application, and illustrate the implementation of basic object-oriented concepts in the Presentation Manager environment.

4.1 Windows

A window is the embodiment of an application object within the Presentation Manager application model. To the end user, a window appears as a rectangular display area on the screen. From an application viewpoint however, the concept of a window is far more powerful than this. Windows may be of two basic types:

- **Display windows** have a physical identity represented by a rectangular area on a screen; in this case the window represents a view of a conceptual display surface known as a **presentation space**, which is in fact a data entity being represented on the screen. This is the average end user's concept of a window.
- **Object windows** have no physical manifestation, and are merely addresses or "handles" to which messages may be directed. An object window is typically associated with an entity such as a file or database, and is used to access this entity and perform actions upon it. The object window concept is very important to the implementation of object-oriented principles under Presentation Manager, since it enables the creation of non-visual objects within the Presentation Manager application model. See 4.6.1.2, "Object Windows" on page 56 for further information.

Windows respond to events, communicated by way of **messages**, which may originate from Presentation Manager as a result of user interaction, or from other windows existing in the system. Messages are routed to and between windows by Presentation Manager on behalf of the application. Windows may interpret and process messages in different ways, in accordance with the concept of polymorphism.

Each window existing in the system has a unique identifier known as its **window handle**, which is assigned by Presentation Manager when the window is created. This handle is used to identify a window in all communication between that window and other parties, such as Presentation Manager or the user. The window handle is specified as the destination address used when passing messages to a window. See 4.2, "Messages" on page 40 for further discussion of Presentation Manager messages. A window is always aware of its own handle, since the handle is part of every message passed to the window. Presentation Manager provides a number of mechanisms by which a window may determine the handles of other windows in the system in order to pass

messages to those windows; these mechanisms are described in 6.6, "Window Communication" on page 87.

4.1.1 Window Classes

Since many windows may be in existence at any time, windows having similar properties and behavior are grouped into **window classes**, with each window belonging to a class being an **instance** of that class. Window classes may be public (defined by Presentation Manager and usable by all applications in the system) or private (defined by the application and accessible only by that application unless an application developer decides otherwise; see 4.5, "Creating Reusable Code" on page 53). Private window classes are registered to Presentation Manager by the application upon initialization of the application.

Presentation Manager maintains control information relating to each window created in the system, including properties such as window text, current size and location, etc. In addition to this information, an application may specify an additional area of storage to be included within the window control block for application data relating to that window. This storage is known as **window words**. The presence and size of window words is determined for each window class at the time the class is registered to Presentation Manager. However, a new storage area is defined for *each* instance of the class, and window words may therefore be used for instance data. Window words typically contain pointers to dynamically-defined application data structures, which in turn contain the instance data.

4.1.2 Window Procedures

Each window class is associated with a **window procedure**, which defines and/or establishes access to data objects and performs processing related to that window class. In object-oriented terms, the window procedure contains the methods to carry out actions upon the data object referenced by the window.

A window procedure is normally invoked by Presentation Manager on behalf of the application. The window procedure interprets messages passed by Presentation Manager to the window and invokes a method (that is, a coherent set of application statements and/or routines) depending on the nature and contents of the message.

See 4.3.2, "Window Procedures" on page 46 for a more complete discussion of window procedures' structure and behavior.

4.2 Messages

All interaction between the user and windows, or between one window and another in the Presentation Manager environment, takes place by way of **messages**. Whenever the user presses a key, moves or clicks the mouse, selects an item from a menu, etc., a message is generated and placed on a system message queue. Presentation Manager takes these messages in the order they were received by the system, determines the window for which each message is intended, and routes the message to a message queue belonging to the application that "owns" that window. The application then dequeues each message in turn, and routes the message via Presentation Manager to the window procedure associated with that window, which processes the message.

Messages may be of three types:

- *User-initiated;* that is, the message is generated as the result of the user selecting an action-bar item, pressing a key, etc.
- *Application-initiated;* that is, the message is generated by one window within the application for the communication of an event or required action to another window.
- *System-initiated;* that is, the message is generated by Presentation Manager as the result of some system event such as a window having been resized or moved.

A Presentation Manager application has the ability to process messages of any of the three types, which allows the application to respond to any type of event, regardless of its origin.

4.2.1 Message Classes

Messages are grouped into **message classes**, with each class representing a particular type of event such as character input, mouse movement, etc. Many message classes are defined by Presentation Manager, and messages of these classes are usually dispatched by Presentation Manager to inform an application of system-initiated events. These system-defined message classes are described, along with the default processing provided by Presentation Manager for each class, in the *IBM OS/2 Version 2.0 Presentation Manager Reference*.

An application developer may define additional message classes unique to his or her particular application, for use by that application. Application-defined messages typically serve as a means of communication between windows, where one window passes information to another window, for processing by the window procedure associated with that window. The destination window may then return a message to the calling window indicating completion, or may pass a message to a third window to trigger an additional action, dependent upon the requirements and design of the application.

For example, the user may elect to update a file by selecting an item from a menu bar. The window procedure associated with the display window to which the menu bar belongs may pass a message to an object window associated with the file to be updated. The window procedure for this window would make the change and then pass a message to a third object window, which logs the update before passing a message back to the original window procedure indicating that the update is complete.

Note that message classes need not be specific to a particular window class; messages of the same class may be passed to different window classes, with different results depending upon the processing of that message by the window procedure belonging to that window class. This is in accordance with the object-oriented principle of polymorphism.

4.2.2 Message Structure

In order to allow any window the ability to process any message class, Presentation Manager defines a standard format for messages. In the Presentation Manager environment, a message is composed of four distinct attributes:

- A **window handle** identifying the window for which the message is intended
- A **message ID** identifying the particular class of message

- Two **message parameters**, which are 32-bit fields containing a variety of information, depending upon the class of message received.

All Presentation Manager applications must contain a message processing loop, which receives the message from Presentation Manager (see 4.3, "Application Structure" on page 43), and routes it, using Presentation Manager message-dispatching functicns, to the appropriate window procedure for processing. Thus Presentation Manager actually invokes the window procedure on the application's behalf.

4.2.2.1 Message Identifier

The message ID identifies the message class to which each message belongs, and is in fact an integer value which is typically replaced by a symbolic name for ease of use. The symbolic names for all system-defined message classes are defined by Presentation Manager; symbolic names for application-defined (private) message classes must be declared by the application developer in the application's source code. This is typically achieved by declaring an integer constant, the value of which is specified as an offset from the system-defined value WM_USER. For example:

```
#define WMP_MYMESSAGE      WM_USER+6
```

The use of an offset to a system-defined constant, rather than an absolute numeric value, eliminates the chance of using the same integer value as a system-defined message class (with consequently unpredictable results), and avoids the necessity to alter application code should the number or definition of system-defined message classes be altered in future versions of Presentation Manager.

4.2.2.2 Message Parameters

As noted above, message parameters may contain a variety of information. When used for communication between window procedures, the window handle of the calling window may be passed to the destination window as one of the message parameters, in order that the destination window procedure may dispatch an acknowledgement or reply message to the calling window. Qualifiers to the message type or small items of data may also be passed; for example, the Presentation Manager-defined WM_COMMAND message class (indicating a menu selection by the user) uses the first message parameter to identify the menu item selected.

When large data structures are required to be passed between window procedures, and the data obviously cannot be contained within the two 32-bit message parameters, the convention is to allocate a memory object for the required data structure using the **DosAllocMem()** function, and to pass a pointer to that memory object as a parameter to the message. Presentation Manager provides a number of macros to enable various types of data to be placed into and retrieved from message parameters. The insertion and retrieval of data into and from message parameters is described in 6.6.6, "Creating Message Parameters" on page 93.

4.2.3 Message Processing

Messages passed to a window may be processed in one of two basic ways:

- *Synchronously*, in which case the message is passed directly to the target window and is processed immediately; the window from which the message originated suspends its execution until the message is processed.
- *Asynchronously*, in which case the message is placed on a queue from which it is subsequently retrieved and passed to the target window; the window from which the message originated continues execution immediately after the message is placed on the queue.

Synchronous and asynchronous processing are described in more detail in 4.3.2.1, "Invoking a Window Procedure" on page 47.

A message may also be dispatched to multiple windows at the same time, using the message broadcasting facilities provided by the Presentation Manager programming interface. Broadcasting may be either synchronous or asynchronous. The implementation of message broadcasting is discussed in more depth in 6.6, "Window Communication" on page 87.

Messages that are not explicitly processed by an application in its window procedures are passed to a default window procedure supplied by Presentation Manager, which contains default processing for all message classes (in the case of application-defined message classes, this default window procedure simply ignores the message). This technique is in accordance with the principle of inheritance, in that a window procedure only contains methods for those message classes with which it is directly concerned, and allows other messages to be processed by the existing default window procedure. Default processing for each message class is described in the *IBM OS/2 Version 2.0 Presentation Manager Reference*.

For message classes that are processed by the application's window procedures, the last operation in the message processing should be to provide a return code to Presentation Manager. In many cases, this return code determines whether Presentation Manager performs its default message processing, after the application's own message processing is complete. The default message processing may or may not be desirable, depending upon application requirements. This ability allows system-initiated events to be easily detected and trapped by a Presentation Manager, enabling the application to perform its own processing for the event before allowing the default processing to occur.

4.3 Application Structure

All Presentation Manager applications have a similar structure. The application is composed of a main routine, which performs initialization and termination functions, and which contains the application's main message processing loop, and a number of window procedures that process messages for window classes created and/or used by the application. These window procedures are invoked and messages passed to them by Presentation Manager on behalf of the application, as shown in Figure 6 on page 44.

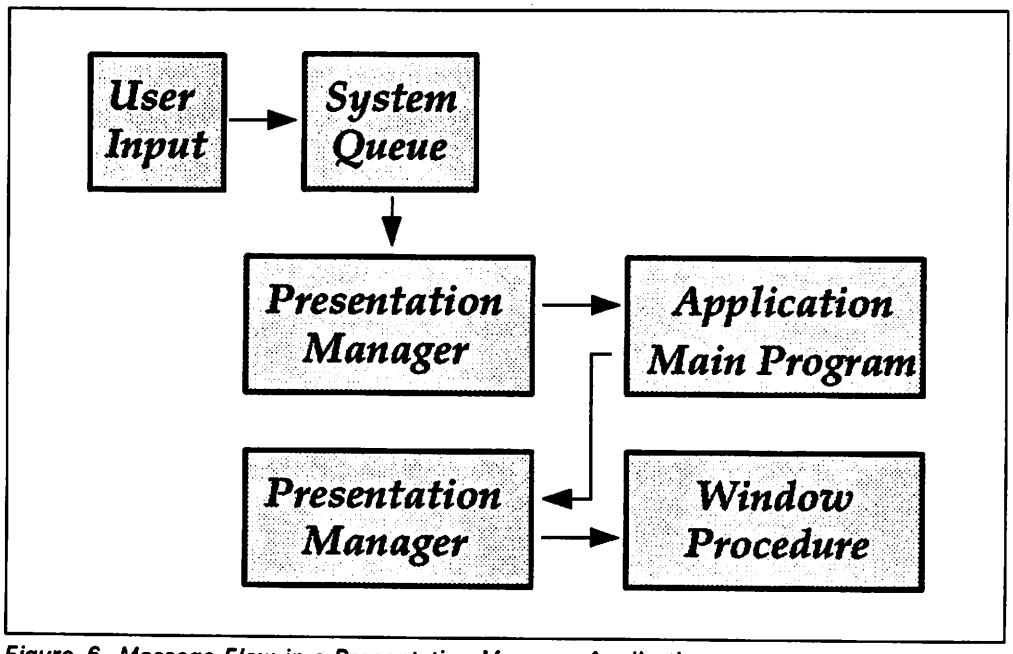

Figure 6. Message Flow in a Presentation Manager Application

Window procedures may also pass messages between one another in order to communicate events. The flow of messages between the window procedures is also controlled by Presentation Manager on behalf of the application.

4.3.1 Main Routine

The main processing routine of a Presentation Manager application performs a number of initialization and termination functions for the application, as shown in Figure 7.

```
int main()
{
    <Global data declarations>

    hAB   = WinInitialize(...);          /* Register application   */
    hMsgQ = WinCreateMsgQueue(...);      /* Create message queue   */

    WinRegisterClass(...);               /* Register window class  */
              :
    hFrame  = WinCreateWindow(...);      /* Create frame window    */
    hClient = WinCreateWindow(...);      /* Create client window   */

    WinAddSwitchEntry(...);              /* Add task manager entry */

    while (WinGetMsg(...))               /* Loop until WM_QUIT     */
          WinDispatchMsg(...);

    WinRemoveSwitchEntry(...);           /* Remove task mgr entry  */

    WinDestroyWindow(hFrame);            /* Destroy main window    */
    WinDestroyMsgQueue(hMsgQ);           /* Destroy message queue  */
    WinTerminate(hAB);                   /* Deregister application */
}
```

Figure 7. Structure of an Application's Main Routine

The application's main routine registers the application to Presentation Manager, creates the application's message queue and defines private window and message classes for the application before actual processing takes place. Other application-specific initialization processing may also take place at this time, such as the definition of global data items. Note however, that the use of global data increases the interdependence of application modules and reduces the potential for subsequent code reuse. Hence global data should be avoided wherever possible.

The main routine also creates the application's main window. Note that from an application viewpoint, a display window is actually a *group* of windows that appear and behave as a single unit. Therefore the frame (with its associated title bar, menu bar, etc.) and the client areas are created separately, as shown in Figure 7. This concept is explained in more detail in 4.6, "Window Hierarchy" on page 54.

During execution of the application, the only function of the main routine is to accept messages from the system queue and route them to window procedures via Presentation Manager. This is performed using a message processing loop, which continues until a message of the system-defined class WM_QUIT is received, at which point the loop terminates and allows the application's termination processing to occur.

Upon termination of the application, the main routine destroys any remaining windows, along with the application's message queue, and deregisters the application before terminating. Any global data items acquired during initialization are also released at this time.

When the user selects "Exit" from the menu bar, or selects the "Close" option on the system menu of the application's main window, the application is *not* automatically terminated. These messages are passed to the window procedure for that window, and may be processed by the application. The typical action performed by the window procedure in such cases is that a message of class WM_QUIT is posted to the application's message queue, which causes the message processing loop to terminate. Conventions for closing windows and terminating applications are discussed further in 6.5.3, "Window Closure" on page 80.

Since Presentation Manager actually *informs* the application that it is being terminated rather than simply shutting the application down, the application is given a chance to perform any necessary termination processing and exit in an orderly manner. When a window is destroyed, it receives a WM_DESTROY message from Presentation Manager, which may be processed by the window procedure to allow orderly termination of processing. This enables the preservation of data integrity by Presentation Manager applications, which need not rely on the user completing a logical unit of work before terminating; any uncompleted units of work may be placed in an interim storage area on disk, or simply backed out as part of the WM_DESTROY message processing.

This feature is also used by the "Shutdown" procedure of the Workplace Shell. When the user selects "Shutdown" from the menu bar, Presentation Manager posts a WM_QUIT message to the main window of each application executing in the system, informing the application that it is being terminated and allowing any termination processing to take place. This facility allows for an orderly shutdown of the system and preserves the integrity of data objects.

4.3.2 Window Procedures

It has already been mentioned that each window class within an application, whether a display window or an object window, is associated with a window procedure, which receives all messages intended for windows within that class. The window procedure contains the methods used to perform actions upon the data object to which the window pertains, and thus contains the application logic for the manipulation of data objects. It may thus be said that window procedures form the "heart" of a Presentation Manager application.

Upon invocation, the window procedure determines the type of message passed to it, and may either process the message explicitly or pass it on to a default window procedure supplied by Presentation Manager for standard processing. A window procedure is essentially an extended CASE statement, as illustrated in Figure 8. Each case within the window procedure contains a set of application statements and/or routines necessary to perform a particular action. Thus in object-oriented terminology, each case is a method in its own right.

```
MRESULT EXPENTRY wpMain(HWND hWnd, ULONG ulMsg, MPARAM mp1, MPARAM mp2)
{
        switch (ulMsg)
                {
                case WM_CREATE:
                     WinDefWindowProc(hWnd, ulMsg, mp1, mp2);
                     <perform initialization of instance variables>
                     <define, create or establish access to data objects>
                     return((MRESULT)FALSE);
                     break;
                case WM_COMMAND:
                     <determine command by examining message parameters>
                     <perform processing for menu command>
                     return((MRESULT)0);
                     break;
                case WM_HELP:
                     <perform help processing>
                     return((MRESULT)0);
                     break;
                       :
                       :
                case WM_DESTROY:
                     <back out any incomplete updates>
                     <close all open data objects>
                     return((MRESULT)0);
                     break;
                default:
                     return((MRESULT)WinDefWindowProc(hWnd,
                                                      ulMsg;
                                                      mp1,
                                                      mp2));
                     break;
                }
}
```

Figure 8. Structure of a Window Procedure

A window procedure is declared to return the type MRESULT, which is a 32-bit data type declaration provided by Presentation Manager, and indicates the

nature of the window procedure's return value. Note that in the above example, the returned values differ depending upon the message class; see 4.3.2.3, "Returning from a Window Procedure" on page 49 for further discussion. Note also that in the default case shown in Figure 8, the window procedure did not decide the return value itself, but used the value returned by the Presentation Manager-supplied default window procedure. This is an established Presentation Manager convention.

A window procedure should always be declared with the *system* linkage convention; this is typically achieved by declaring the function to be of type EXPENTRY. This type identifier is defined in the system header file *OS2DEF.H*, and simply specifies use of the *system* linkage convention. Use of the *system* linkage convention is required since a window procedure is invoked by Presentation Manager on the application's behalf, rather than directly by the application. Note that under previous versions of OS/2, the EXPENTRY keyword resulted in use of the *pascal* linkage convention, for precisely the same reason.

Subject to programming language conventions, a window procedure has the ability to define a data object or instance data, or to establish access to an existing data object as part of its initialization processing. When a window is created by Presentation Manager in response to an application's request, Presentation Manager immediately dispatches a message of the system-defined class WM_CREATE to that window (see Figure 8). The window procedure may process this message in order to define instance data or establish access to data objects. Typically, a window procedure requests the allocation of a memory object as a control block for its instance data. Initial values for the instance data are then placed in the control block, and a pointer to the control block is stored in the window words. A window procedure thus supports the object-oriented concept of encapsulation, by allowing the dynamic allocation of and establishment of access to data objects, within a single application object.

Note that prior to allocating instance data or performing any other processing for the WM_CREATE message, a window procedure should invoke the default message processing provided by Presentation Manager in the **WinDefWindowProc()** function. This ensures that the initialization of Presentation Manager's control data for the window is completed prior to WM_CREATE processing by the window procedure. This in turn ensures that the window handle, window words, etc., will be available during the window procedure's WM_CREATE processing.

4.3.2.1 Invoking a Window Procedure

A window procedure is invoked by dispatching a message to a window of the class with which the window procedure is associated. Messages passed to a window are typically initiated as the result of user interaction or application events, or by Presentation Manager to indicate system events.

While window procedures may be invoked directly using a normal subroutine call, it is recommended that messages be used for *all* communication between window procedures. This conforms to standard object-oriented practice, in that an object should be accessible only via a message passed to it.

Messages may be used to invoke a window procedure in two ways:

- A message may be *sent* directly to the window procedure using the **WinSendMsg()** call, in which case the window procedure executes synchronously, and control does not return to the calling procedure until

processing is completed. This method of invocation is similar to a normal subroutine call, but preserves the message-driven structure of the application. Note that since the message is sent directly to the window procedure and not placed on a queue, the normal serialization of message processing is disturbed. This may cause different results from those intended by the user; thus **WinSendMsg()** should be used with care.

- A window procedure may also be invoked by *posting* a message to a queue associated with the window procedure, using the **WinPostMsg()** call. With this call the window procedure executes asynchronously, and control returns to the calling procedure immediately after the message is placed in the queue. This method of invocation provides a convenient and powerful means for serialized and yet asynchronous processing. It then becomes the responsibility of the application developer to ensure synchronization between different window procedures.

Invoking a window procedure by posting a message to it via a queue has an advantage over the use of **WinSendMsg()** or a direct subroutine call in that, where multiple windows are passing messages to a single receiving window, these messages are queued by Presentation Manager and dispatched to the receiving object in the order in which they were initiated. Provided all sending windows obey the established conventions and post messages to queues, this ensures the correct sequencing of message processing by the receiving window, helps preserve the user's intention and facilitates maintaining the integrity of data objects.

A window procedure accepts four parameters upon invocation; these correspond to the four attributes of a message as described in 4.2, "Messages" on page 40 and to the parameters of the **WinSendMsg()** and **WinPostMsg()** functions, and are as follows:

1. The handle of the window for which the message was intended
2. A message-class identifier
3. Two 32-bit message parameters.

Note that the behavior of a window procedure, and the result obtained from processing by a window procedure, are dependent upon the class and contents of the message sent to it. Similarly, the same message class may be interpreted in a different manner by window procedures belonging to different window classes. In this way, a window procedure supports the object-oriented concept of polymorphism.

4.3.2.2 Window Procedure Processing

A window procedure will normally determine the message class, and take action based upon that class and the contents of the message parameters. Where the action involves the manipulation of data objects and/or instance data, the window procedure typically obtains access to the window's control block by retrieving its pointer from the window words. The window procedure then has access to the data values, resource handles, etc., required to complete the action.

Note that the example given earlier in this chapter contains explicit processing options for only four types of messages; the application allows Presentation Manager to handle all other types of messages, by passing the message to the system-supplied default window procedure using the **WinDefWindowProc()**

function. This illustrates one of the basic principles of a Presentation Manager application; the window procedure processes *only* those messages with which it is explicitly concerned, and passes *all* other messages to Presentation Manager for default processing. A window procedure *must* pass such messages on to Presentation Manager, or unpredictable results may occur.

This "catchall" approach to implementation also allows the stepwise implementation of methods during application development. An application developer may code a window procedure such that all command messages are, by default, passed to a routine that displays a message informing the user that the requested action is not yet implemented. As the method for each action is designed and coded, a case for that action may be added to the window procedure. Thus implementation of methods for an object proceeds in a stepwise and independent fashion until all necessary methods are implemented.

As mentioned earlier in this chapter and illustrated in Figure 8, a window procedure may process messages of any type, including the WM_DESTROY message, which is posted to a window upon termination. A window procedure may explicitly process this message in order to close files and terminate access to data objects in an orderly manner, thus preserving the integrity of those data objects. This ability allows the window procedure to further support the principle of encapsulation.

4.3.2.3 Returning from a Window Procedure

By convention, a window procedure typically returns a Boolean value (type MRESULT) to its caller, to indicate the result of message processing for that message. The value returned is significant, since Presentation Manager takes action depending upon that value. The defined return values for each system-defined message class, along with the default processing provided by Presentation Manager for each class, are given in the *IBM OS/2 Version 2.0 Presentation Manager Reference*. Naturally, the return values for user-defined message classes are defined by the application developer.

If the window procedure has been invoked synchronously using **WinSendMsg()**, the result is returned by Presentation Manager to the calling window procedure, which may interrogate and act upon it. If the window procedure has been invoked asynchronously using the **WinPostMsg()** function, the result is returned to Presentation Manager *only*. Presentation Manager then uses this result to determine whether any post-processing should be carried out for the message. Note that while the **WinPostMsg()** function call also provides a Boolean return code to the caller, this code only indicates whether the message was successfully posted to the queue, and *not* the successful processing of the message by the target window procedure.

4.3.3 Dialog Procedures

A dialog procedure is a special type of window procedure that is associated with a modal dialog box and processes messages intended for that dialog box. The structure of a dialog procedure is similar to that of a "normal" window procedure, and it processes messages in the same way, although certain message classes received by a dialog procedure are different from those received and processed by a normal window procedure. The structure of a typical dialog procedure is shown in Figure 9 on page 50.

```
MRESULT EXPENTRY dpDProc(HWND hWnd, ULONG ulMsg, MPARAM mp1, MPARAM mp2)
{
        switch (ulMsg)
                {
                case WM_INITDLG:
                     <extract initialization data from message parameters>
                     <establish initial values for control window data>
                     return((MRESULT)TRUE);
                     break;
                case WM_CONTROL:
                     <determine originator of message and message contents>
                     <perform control-specific processing>
                     return((MRESULT)0);
                     break;
                case WM_COMMAND:
                     switch LOUSHORT(mp1):
                            {
                            case DID_OK:
                                 <complete dialog>
                                 WinDismissDlg(hWnd,TRUE);
                                 break;
                            case DID_CANCEL:
                                 <cancel dialog>
                                 WinDismissDlg(hWnd,FALSE);
                                 break;
                            }
                     return((MRESULT)0);
                     break;
                default:
                     return((MRESULT)WinDefDlgProc(hWnd,
                                                   ulMsg,
                                                   mp1,
                                                   mp2));
                     break;
                }
}
```

Figure 9. Structure of a Dialog Procedure

Since modal dialog boxes do not belong to a class, they are not registered to Presentation Manager in the manner of a normal window. A dialog procedure is associated with a dialog box as part of a **WinDlgBox()** call, which loads a modal dialog box and processes the dialog as a single operation, or a **WinLoadDlg()** call, which loads the dialog box into memory but does not process it; the dialog box may subsequently be processed by a **WinProcessDlg()** call. The processes of providing input to and obtaining results from a dialog procedure are discussed in Chapter 6, "Building a Presentation Manager Application."

Note that since a modeless dialog box is simply an optimized (non-sizable) window with no other inherent properties such as modality, it receives the same messages as a standard window, and its methods are therefore contained within a normal window procedure rather than a dialog procedure.

Note that a dialog box must use the *system* linkage convention, since it is simply a specialized form of window procedure. This is achieved using the EXPENTRY keyword.

Upon creation, a dialog box receives a message of the system-defined class WM_INITDLG. This is similar to the WM_CREATE message received by a normal window upon creation, and may be processed in a similar way. The first parameter in the WM_INITDLG message may be used to pass a pointer to the dialog procedure, referencing a data structure containing initialization information or other application-specified data.

A dialog box also receives messages of class WM_CONTROL indicating events occurring in control windows within the dialog box. The window identifier of the control window that dispatched the message, along with the nature of the message, is indicated in the message parameters. The WM_CONTROL message is described in the *IBM OS/2 Version 2.0 Presentation Manager Reference*. The dialog procedure may wish to explicitly process events indicated by WM_CONTROL messages, or it may allow such messages to pass on to the default Presentation Manager-supplied dialog procedure **WinDefDlgProc()**.

A dialog box also typically receives WM_COMMAND messages, which are generated when a pushbutton within the dialog box is pressed by the user. The identity of the pushbutton is indicated in the first parameter of the WM_COMMAND message.

The symbolic names DID_OK and DID_CANCEL are defined by Presentation Manager, and by convention are used to refer to the "OK" and "Cancel" pushbuttons respectively. The definition of dialog boxes is described in detail in Chapter 9, "Presentation Manager Resources."

A dialog procedure is terminated and the dialog box is destroyed when the dialog procedure issues a **WinDismissDlg()** call to Presentation Manager, typically as a result of the user pressing a pushbutton. By convention, this call specifies a parameter indicating TRUE if the user completed the dialog by pressing an "Enter" or "OK" pushbutton, or FALSE if the user cancelled the dialog by pressing a "Cancel" pushbutton. The value specified in this parameter is returned to the window procedure that issued the **WinDlgBox()** or **WinProcessDlg()** call, as the return value from that call.

Message boxes do not require an application-supplied procedure to carry out their processing. The simple nature of the message box dialog allows it to be processed by Presentation Manager on the application's behalf, and to return the result to the application for subsequent action.

4.3.4 Subroutines

Subroutines may be used in a Presentation Manager application, in the same way as they are used in any other application. However, in order to conform to object-oriented practices, subroutine calls should only be used to achieve functional isolation within the methods of a single application object, or to perform a standard processing function that is common to a number of objects; in this case, the scope of each execution instance of the subroutine is limited to a single object. In accordance with object-oriented programming guidelines, communication between objects (windows) should be achieved using messages, and hence window procedures should not be invoked directly as subroutines. See 6.7, "Passing Control" on page 95 for further discussion on the use of subroutines in Presentation Manager applications.

4.3.5 Partitioning the Application

With the dynamic linking capabilities of OS/2, it is possible to partition an application into a number of executable modules. A single base program may be augmented by one or more dynamic link libraries. Such an approach has a number of advantages:

- Application code that is only executed in exceptional circumstances, such as little-used functions, exception and error-handling routines etc, is not loaded unless it is required. This may significantly reduce the load time and memory requirements of the application.
- Common functions may be shared between applications, since dynamic link libraries are re-entrant and a single memory-resident copy of the library code may be used by all applications. This can further reduce the memory requirements of an application.
- Functions placed in dynamic link libraries are isolated from the main application, and may be modified without the need to re-link the application. This facilitates application maintenance and update, since only the new version of the DLL need be distributed.

Applications written for the Workplace Shell should be partitioned in this manner. Since the shell displays all objects on the desktop at system initialization, all applications must potentially be loaded at this time. This can dramatically increase the time required for system initialization, along with the overall memory requirements of the system.

These requirements may be significantly reduced by having only the minimum code (that is, the code required to accept and identify messages) loaded at application startup, and placing all processing function into dynamic link libraries that are loaded only when one of their entry points is called.

Application developers must give careful consideration to correct partitioning of the application. Groups of functions that are interdependent and which call one another, or are typically called in close sequence should be placed in a single DLL module. Functions that are independent of one another should be placed in separate DLLs. This approach will minimize the load time and memory requirements of the application.

4.4 Presentation Manager Resources

Presentation Manager allows the application developer to define application **resources** externally to the application code. Resources may include definitions for the following items:

Fonts Graphic fonts may be created and modified using the **Font Editor** supplied as part of the *IBM Developer's Toolkit for OS/2 2.0.*

Icons Application and window icons, mouse-pointers and bitmaps may be created and modified using the **Icon Editor** supplied as part of the *IBM Developer's Toolkit for OS/2 2.0.*

Menus Menu bars and pulldown menus may be defined for display windows.

String Tables Tables of text strings may be defined for use by an application.

Accelerator Tables Tables of accelerator keys (for example, F3 for Quit) may be defined for display windows.

Help Tables Tables of help panels may be defined for each display window or each control window in a dialog box. See Chapter 15, "Adding Online Help and Documentation" for further discussion of help panels.

Dialog Templates Dialog boxes may be created or modified and stored as dialog templates, using the **Dialog Box Editor** supplied as part of the *IBM Developer's Toolkit for OS/2 2.0.*

Window Templates Window templates may be created or modified and stored as window templates, using the dialog editor supplied as part of the *IBM Developer's Toolkit for OS/2 2.0.*

Except where noted above, resources are defined in a **resource script file**, an ASCII text file that may be manipulated using a standard text editor. This resource script file serves as input to a **resource compiler**, which is provided as part of the *IBM Developer's Toolkit for OS/2 2.0.* The resource compiler produces a precompiled version of the resources, which is then incorporated into the application's executable code or stored in a dynamic link library for use by one or more applications.

It is usual for simple text-based resources such as menus and string tables to be placed directly into the resource script file using an ASCII text editor. However, non-textual resources such as icons or bitmaps, or more complex text-based resources such as dialog templates, are typically stored in separate files and referenced from the resource script file.

A major benefit of defining such resources externally to the application is that changes may be made to resource definitions without affecting the application code itself. Modifications such as new icons, altered commands or menus, etc., may be implemented quickly and easily by making simple changes at a single point in the application.

As a further example, the task of providing national language versions of an application is simplified, since all text such as menus and messages may be defined outside the application code, and multiple language-specific versions of such resources may be linked into a single version of the application code. In this way, the user interface properties of a display window may be modified without affecting the internal implementation of the window procedure or its methods.

The creation and use of Presentation Manager resources is discussed in Chapter 9, "Presentation Manager Resources."

4.5 Creating Reusable Code

The ability to define resources such as window and dialog templates externally to the application, in conjunction with the dynamic linking facilities provided by the OS/2 operating system, provides a powerful tool for the creation of generic application objects, comprised of window/dialog templates and their associated window or dialog procedures. These application objects may be defined and

stored in dynamic link libraries for subsequent use by one or more applications. This practice is in accordance with the guidelines for Systems Application Architecture Common Applications, which provide for common application services within and between environments, as well as common user applications. This concept is further discussed in Chapter 17, "Generic Application Modules." Note however, that the use of generic application objects presupposes that the nature of and message interfaces to such application objects are well-defined and documented, in order to allow application developers to correctly select and interact with the generic objects.

For instance, a standard dialog box that will be used by many applications could be defined in a dialog template, resource compiled and stored in a dynamic link library, along with the dialog procedure which performs the processing for that dialog. The dialog can then be loaded from the DLL by any application which needs to use it. The dialog need be coded only once, and may be modified at any time while requiring no source code changes to the applications that access it. The fact that DLL code is not bound with the application at link edit time like other library code also means that no changes are required to the object code of the applications, and thus recompilation and link edit is not required. An example of this technique is given in Chapter 9, "Presentation Manager Resources."

The window or dialog procedure associated with a generic object should contain all the methods normally used to perform actions upon that object, but need not contain every action that will ever be necessary. If an application requires a specialized action on a generic object (that is, a previously undefined action or a modification of an existing action), the window acting as a handle to that object may be subclassed, and a new window procedure substituted for the existing window procedure. This new window procedure would contain methods to process the specific messages in which it has an interest, and would then invoke the original window procedure to handle any other message classes, in accordance with the object-oriented principle of inheritance. Subclassing is discussed further in 4.7, "Subclassing" on page 57.

4.6 Window Hierarchy

Presentation Manager organizes windows hierarchically; each window in the system has a parent, and may optionally have an owner. These parent and owner relationships determine the behavior of the window in response to certain messages, and may be used by applications to navigate the window hierarchy.

4.6.1 Parent/Child Relationship

The parent/child relationship between windows is mentioned in *OS/2 Version 2.0 - Volume 3: Presentation Manager and Workplace Shell*, with regard to the clipping of a child window to the borders of its parent. However, this hierarchy goes further in Presentation Manager, since *all* windows, both display windows and object windows, have a designated parent window. For top-level display windows, this parent window is the desktop, and is identified by the HWND_DESKTOP constant. Other display windows within an application, which are child windows of the application's main window, may have the top-level application window as their parent, or indeed subsequent levels of the window hierarchy may be created, dependent on application requirements. A window's parent is identified to Presentation Manager by the application when the window

is created. Thus the window hierarchy within a particular desktop is dynamically defined at execution time.

As well as being uniquely identified by its window handle, a child window may also be identified by a window identifier, which is unique between children of a particular parent window. This identifier is an integer value, which in practice is usually replaced by a more meaningful symbolic name that defines an integer constant. The window identifier is supplied as a parameter when the application requests creation of the window by Presentation Manager. When a window's parent and identifier are known, the **WinWindowFromID()** function may be used to determine its window handle so that operations may be performed upon it. See 6.6, "Window Communication" on page 87 for further information.

The parent/child hierarchy is useful for application design purposes, since in many cases, a window and its children may be regarded and manipulated as a single unit. For example, sizing a parent window automatically clips all children of that window to the boundaries of the parent, and closing a parent window results in each of its children being closed. This simplifies the application logic required for applications that create multiple windows.

4.6.1.1 Frame and Client Windows

The concepts of frame and client areas for a window are discussed in *OS/2 Version 2.0 - Volume 3: Presentation Manager and Workplace Shell*. In fact, these frame and client areas are separate windows in their own right; the frame window of the application's "main window" is a top-level window with the desktop as its parent, and the client window is a child of the frame window. Frame control windows such as maximize/minimize icons, the title bar, the menu bar, etc., are also separate windows from the application viewpoint, and are regarded as children of the frame window. Note that although they are separate windows, the end user perceives and manipulates the entire group as a single unit.

The frame window and its children all belong to system-defined generic window classes, and thus have their own window procedures defined by Presentation Manager. The exception is the client window, the window class of which is defined by the application; the window procedure is therefore defined to Presentation Manager when the window class is registered. Note that window procedures for system-defined window classes may be subclassed by the application in order to provide specialized processing of certain messages.

The children of a frame window have specific window identifiers assigned to them by Presentation Manager. These window identifiers are unique for each frame window. The predefined window identifiers are shown in Table 1.

Table 1 (Page 1 of 2). Window Identifiers. This table shows the window identifiers assigned by Presentation Manager to the children of a frame window.

Child Window	Window Identifier	Defined by
Client Window	FID_CLIENT	Application
System Menu	FID_SYSMENU	Presentation Manager
Menu Bar	FID_MENU	Presentation Manager
Title Bar	FID_TITLEBAR	Presentation Manager
Min/Max Icon	FID_MINMAX	Presentation Manager
Vertical Scroll Bar	FID_VERTSCROLL	Presentation Manager

Table 1 (Page 2 of 2). Window Identifiers. This table shows the window identifiers assigned by Presentation Manager to the children of a frame window.		
Child Window	**Window Identifier**	**Defined by**
Horizontal Scroll Bar	FID_HORZSCROLL	Presentation Manager
Note: The "Defined by" column indicates whether the window procedure that determines a window's appearance and behavior is supplied by Presentation Manager or the application.		

These identifiers may be used to communicate with child windows of a particular frame window, without the necessity to determine the window handle of the child window. This concept is applicable to all windows including control windows within a dialog box. See 6.6, "Window Communication" on page 87 for further information.

4.6.1.2 Object Windows

Object windows do not have a parent in the visual sense, as considerations such as clipping do not arise (since the window is never displayed). For the purposes of standardization, Presentation Manager considers every object window to have a conceptual parent; this parent may be referenced using the constant HWND_OBJECT. This technique allows an object window to be created using the same Presentation Manager function as that used to create a display window. It is also useful in allowing logical grouping of windows with similar functions, or which need to be treated as a group for application purposes, under a single conceptual parent.

For example, all the object windows created by a particular application may be grouped as children of a single "dummy" parent window. When the application terminates and wishes to destroy all these windows, only a single function call to destroy the parent need be issued; Presentation Manager will automatically destroy each of the children in turn before destroying the parent. Due to the way in which Presentation Manager destroys windows by first passing a WM_DESTROY message to the window, each object window is given a chance to exit in an orderly manner.

4.6.2 Window Ownership

A window may have an *owner* as well as having a parent. While a window's relationship to its parent is mainly concerned with display, the relationship with its owner is concerned primarily with function. The owner is assumed to have some interest in events that take place within a window. For example, a frame window is the parent of its frame control windows (icons, menu bar, etc.,) and is also their owner. When certain events take place, such as the user selecting an item from a menu bar, the control windows notify their owner by dispatching notification messages (the menu bar for instance, typically dispatches a message of class WM_COMMAND). Thus the owner receives notification of the event, and may perform some action in response.

The concept of ownership is usually applied to system-defined window classes such as control windows. Since the window procedures for these window classes are defined by Presentation Manager rather than by the application, it is necessary for a control window to notify its owner of any event that may be significant to the owner.

There is typically no owner relationship between frame and client windows. Messages received by the frame window that are deemed to be of possible interest to the client window are passed to the client window automatically by the system-supplied frame window procedure, by virtue of the predefined parent/child relationship between the frame and client windows.

Should application requirements dictate, the application developer may establish an owner relationship between any two windows within his/her application, provided those windows are created by the same thread. Owner relationships are not permitted between windows created by different threads.

4.6.3 Z-Order

A desktop is typically regarded as a two-dimensional display space; in fact, a desktop is three-dimensional, since when windows overlap on the desktop, one window is conceptually "on top of" the other. This concept of "stacking" windows applies even when windows do not actually overlay one another on the desktop. The order in which windows appear on the desktop is known as the **z-order**. The z-order is known to Presentation Manager, and a number of function calls are provided that enable an application to request window handles of specific windows in the z-order using the **WinGetNextWindow()** function to obtain the handle of the next window in the z-order, or the **WinQueryWindow()** function to obtain the handle of a window at a specified position in the z-order.

Since the z-order changes dynamically as different applications create and destroy windows, Presentation Manager takes a "snapshot" of the desktop state when the application issues a **WinBeginEnumWindows()** call. This function accepts a window handle as a parameter, and the z-order of all immediate children of that window is recorded by Presentation Manager. A call to this function should be issued before any **WinGetNextWindow()** call is issued, or before any **WinQueryWindow()** call is issued that specifies a position in the z-order. When the application no longer wishes to interrogate the recorded window hierarchy, a **WinEndEnumWindows()** call should be made.

The concepts of z-order and window enumeration are useful in circumstances where an operation or sequence of operations must be performed on a number of windows in order. Windows with the same parent always appear contiguously in the z-order, and thus may be easily processed in succession.

4.7 Subclassing

Windows may be subclassed by replacing the window procedure defined by the window class with another window procedure. This new subclass window procedure typically processes some of the messages routed to it, and then calls the original window procedure to process any other messages. This technique parallels the definition of subclassing given in Chapter 3, "Object-Oriented Applications."

Presentation Manager implements subclassing by having the application call the **WinSubclassWindow()** function, specifying the handle of the window to be subclassed and the new subclass window procedure. Note that only the window specified is affected by the **WinSubClassWindow()** function call; other windows of the same class are not subclassed. Once the call is successfully issued, any messages destined for the original window procedure are automatically routed to the subclass window procedure by Presentation Manager. The object

(whether a window or the operating system) from which the message originated is unaware of the subclass window procedure's interference. An example of a subclass window procedure is given in Figure 25 on page 86.

The subclassing concept enables messages destined for particular windows to be intercepted and the processing resulting from certain messages to be altered. This provides a powerful mechanism that facilitates the creation and use of generic windows (application objects), while retaining the ability for the application to modify the behavior of such windows should the need arise. The use of subclassing enables a newly created window to take on the properties and methods of existing window classes in accordance with the principle of inheritance.

Presentation Manager enables the effect of subclassing a window to be reversed by the application issuing the **WinSubclassWindow()** call a second time for the same window, specifying the original window procedure. Presentation Manager then routes messages directly to their intended destination. This capability allows windows to be *temporarily* subclassed to meet changing requirements at different points during application execution.

4.8 Summary

It can be seen from the foregoing discussion that Presentation Manager provides a base that facilitates the implementation of module-based object-oriented conventions by application programs. The concepts of an application object and its methods are implemented under Presentation Manager as a window and its window procedure.

Windows are grouped into classes and a window procedure is associated with a window on the basis of its class, in a parallel to the concept of allocating methods to an object class rather than to individual instances of that class. Window classes are defined in isolation however, and the concept of an inheritance hierarchy is not imposed by Presentation Manager, thus further enhancing the potential for efficient reuse by increasing object granularity. Presentation Manager allows windows to be subclassed, in order to allow additional or modified methods to be applied to an object in response to new or specialized actions. This provides an additional enhancement to the capability of code reuse, since it is not necessary to create a new object class in order to implement small modifications to an existing class.

Windows communicate with the system and with each other by way of messages, which are queued and routed by Presentation Manager, and which are processed in a serial fashion by the target window procedure. This messaging model is a practical implementation of the message-driven communication precept of object-oriented application design.

While not supported explicitly by Presentation Manager, the object-oriented concept of encapsulation is supported implicitly by the ability of a window procedure to define and thus "own" a data object. The concept of polymorphism is also supported by Presentation Manager, since the behavior and results obtained from a window procedure are dependent upon, and only upon the class and contents of messages sent to that window procedure. In a similar fashion, the result of a message is dependent upon the window procedure (application object) to which it is passed. The isolation of data objects within an application

object facilitates the containment of change by enhancing application modularity, thus easing the task of change management and application maintenance.

Table 2. Application Object/Window Correlation

Application Object	Supported	Implementation
Message Communication	Yes	PM Message
Class Association	Yes	Window Class
Class Data	Yes	Defined in Window Procedure
Instance Data	Yes	Stored in Window Words
Encapsulation	Yes	In Window Procedure
Polymorphism	Yes	In Window Procedure
Inheritance	Partial	Via Subclassing

The ability to encapsulate the definitions of data objects with the methods used to manipulate those objects, and to store the resulting application objects in object libraries, facilitates the notion of reusability, which is one of the central precepts of object-oriented programming. The dynamic linking facilities provided by OS/2 further extend the potential for reusable application objects. Reusable objects may be defined and stored for use by multiple applications; indeed, multiple objects may direct messages to a single instance of an object executing in the system. The message queueing and serialization provided by Presentation Manager ensures the correct sequence of processing to preserve the user's intention and facilitate the integrity of data objects.

It may be seen that the concept of an application object as defined in Chapter 3, "Object-Oriented Applications" and the implementation of a window under Presentation Manager have a strong correlation. A window may be regarded as the identity of an application object. That object is associated with a data object and a set of methods (the window procedure) that perform actions upon the data object. Class-specific data is defined within the window procedure, while storage for instance data is defined dynamically and pointers typically stored in window words. Windows communicate with the user and with one another by way of messages. Thus the window is the implementation of an application object under Presentation Manager.

Hence Presentation Manager provides an execution environment and a basic application architecture that supports the implementation of object-oriented applications, within the boundaries of IBM Systems Application Architecture. Although it does not provide a complete development environment that enforces object-oriented guidelines, it offers the basis upon which such a development environment may be based.

Chapter 5. The Flat Memory Model

The task of dynamically allocating memory within an application is greatly simplified in the 32-bit OS/2 Version 2.0 environment through use of the flat memory model. The application developer need no longer be concerned with the maximum 64KB segment size imposed by the 80286 processor architecture. Larger amounts of memory may be allocated and subsequently manipulated as single units known as **memory objects**, rather than as multiple segments as was the case with previous versions of OS/2. This reduces application complexity, facilitating improved performance and reducing application development time.

This chapter describes the use of the flat memory model for application programming, in order to allocate and manipulate system memory. The chapter also examines the facilities provided by OS/2 Version 2.0 that enable applications to handle memory protection exceptions.

The concept of the flat memory model is described in *OS/2 Version 2.0 - Volume 1: Control Program*. The functions necessary to manipulate memory from within applications are described in detail in the *IBM OS/2 Version 2.0 Control Program Reference*.

5.1 DosAllocMem() Function

The **DosAllocSeg()** function implemented under previous versions of OS/2 is replaced in Version 2.0 by the **DosAllocMem()** function, which allows allocation of memory objects greater than 64KB in size. To take an example, Figure 10 shows the code necessary under OS/2 Version 1.3 to allocate a 72KB area of memory for use by an application:

```
SEL    sel1, sel2;
PVOID  pSegment1, pSegment2;

DosAllocSeg(0, &sel1, SEG_NONSHARED);
DosAllocSeg(8192, &sel2, SEG_NONSHARED);

pSegment1=MAKEP(sel1, 0);
pSegment2=MAKEP(sel2, 0);
```

Figure 10. Allocating Memory in Previous Versions of OS/2. This example shows the use of the DosAllocSeg() function to allocate multiple segments in order to access 72KB of memory.

The application must then use *pSegment1* to reference the lower 64KB and *pSegment2* to reference the upper 8KB of the memory object. This requires conditional testing for each memory reference, and thereby introduces additional complication to the application code. Use of the **DosAllocHuge()** function simplifies this slightly, but arithmetic is still required in order to correctly calculate offsets within the higher area of memory.

Under OS/2 Version 2.0, a single **DosAllocMem()** function call is required in order to perform the same task, as shown in Figure 11 on page 62.

```
PVOID  pObject;            /* 32-bit linear pointer to memory object */

DosAllocMem(&pObject,      /* Allocate memory object                 */
            73727,         /* Size of memory object                  */
            PAG_READ |     /* Allow read access                      */
            PAG_WRITE);    /* Allow write access                     */
```

Figure 11. Allocating Memory in OS/2 Version 2.0. This example shows the use of the DosAllocMem() function to allocate a single 72KB memory object.

Subsequent references to this memory object may simply use a 32-bit offset within the allocated address range.

Note that since OS/2 Version 2.0 uses paged memory internally, memory allocated using **DosAllocMem()** is *always* allocated in multiples of 4KB. Thus, a request for 10 bytes will actually result in a full 4KB page being committed in real storage. Since this will lead to high fragmentation and consequent waste of memory, the allocation of many small memory objects using **DosAllocMem()** directly is not recommended. Application developers should initially use **DosAllocMem()** to allocate the maximum storage likely to be required, and then use the **DosSubAlloc()** function to allocate individual memory objects. This technique allows the storage of multiple small memory objects within the same 4KB page, thereby reducing fragmentation and making more efficient use of storage.

Note that the **DosAllocHuge()** function provided under previous versions of OS/2 has no counterpart under Version 2.0. This function is not required since **DosAllocMem()** allows the theoretical allocation of memory objects of a size up to that of the application's entire process address space.

Memory objects allocated using **DosAllocMem()** may be freed using the **DosFreeMem()** function.

5.2 Allocating versus Committing Memory

Under OS/2 Version 2.0, there is a distinction between *allocating* a memory object and *committing* that object. This distinction was not present in previous versions of OS/2, and is a very important concept for the application developer to grasp. When a memory object is allocated, space is reserved in the linear address space, but no real storage or swap file space is reserved for the object. This space is only reserved when the memory object or parts thereof are committed. A memory object that has not been committed is known as a **sparse object**.

A memory object may be committed in two ways:

- It may be committed (in its entirety) at the time it is allocated, using the PAG_COMMIT flag in the **DosAllocMem()** function.
- It may be committed in stages at some later point, using the **DosSetMem()** function.

The former technique is intended for small memory objects, the size of which is fixed and can be determined in advance by the application developer. The latter technique is intended for memory objects such as external data files, which may vary in size.

Memory must be committed prior to being accessed by the application. Failure to do this will result in a page fault (Trap 000E) exception.

5.2.1 Committing Storage at Allocation

For memory objects that have a fixed size, such as internal application storage, control blocks and most instance data, memory objects should be committed immediately upon allocation, allowing the application to access the memory object without the inconvenience and additional overhead of explicitly committing the storage at a later time.

Storage for a memory object may be committed using the PAG_COMMIT flag in the **DosAllocMem()** function call used to allocate the memory object, as shown in Figure 12.

```
PVOID  pObject;              /* 32-bit linear pointer to memory object */

DosAllocMem(&pObject,        /* Allocate memory object                 */
            73727,           /* Size of memory object                  */
            PAG_READ   |     /* Allow read access                      */
            PAG_WRITE  |     /* Allow write access                     */
            PAG_COMMIT);     /* Commit storage immediately             */
```

Figure 12. Committing Storage During Allocation. This example shows the use of the PAG_COMMIT flag with the DosAllocMem() function.

The above example creates a 72KB memory object in a similar manner to that shown in Figure 11 on page 62, but commits the storage during allocation, so that is immediately available for use by the application.

5.2.2 Dynamically Committing Storage

Under DOS and previous versions of OS/2, it is common for an application to allocate a small memory segment to contain a data structure. If the data structure outgrows the size of the segment, the segment size may be progressively increased using the **DosReallocSeg()** or **DosReallocHuge()** functions, moving the segments within the machine's physical memory in order to accommodate the increased size requirements. This is not possible under Version 2.0, since the paged memory implementation does not allow memory objects to be moved within memory once they are allocated; hence the **DosReallocSeg()** and **DosReallocHuge()** functions have no counterparts in the 32-bit environment.

Under OS/2 Version 2.0, an application can allocate an area of storage in its process address space, but may commit only a small amount of that storage at the time the application is initialized. In this way, the application does not use a large amount of storage in a situation where it is not required, and thereby avoids placing unnecessary resource demands on the operating system. This can result in improved overall system performance.

If the storage requirements for a memory object increase during execution (for example, the size of a spreadsheet increases), and exceed the amount of storage initially committed, the application may dynamically commit additional storage up to the maximum specified in the **DosAllocMem()** function call that allocated the memory object.

This dynamic commitment of storage is typically achieved using the **guard page technique**. A page within the memory object may be specified as a *guard page* using the PAG_GUARD flag in the **DosAllocMem()** function call or in a **DosSetMem()** call made subsequent to the allocation. Once this is done, any memory reference to that page will generate a **guard page exception**. The guard page exception warns the application that the upper boundary of the committed portion of a memory object has been reached, and allows appropriate action to be taken in order to avoid a page fault exception.

Note that the memory protection scheme implemented by OS/2 Version 2.0 allocates pages to individual processes. An exception is *only* generated when an application attempts to write into a page which is not allocated to the current process under which the application is running. If the page is allocated to the current process, no exception is generated. Use of the guard page technique is therefore strongly recommended in circumstances where the amount of data to be written into a memory object is variable, or where the size of the memory object or its data may grow during execution.

The recommended method of using guard pages is to initially allocate the memory object as a sparse object, and then commit the amount of storage required for the current size of the data, flagging the uppermost page of the memory object as a guard page. This technique is shown in Figure 13.

```
PVOID  pObject;                /* 32-bit linear pointer to memory object */

DosAllocMem(&pObject,          /* Allocate memory object                 */
            73727,             /* Size of memory object                  */
            PAG_READ |         /* Allow read access                      */
            PAG_WRITE);        /* Allow write access                     */

DosSetMem(pObject,             /* Set memory attributes for object       */
          8192L,               /* Two pages (8192 bytes)                 */
          PAG_DEFAULT |        /* Default attributes from allocation     */
          PAG_COMMIT);         /* Commit page                            */

DosSetMem(pObject+4096,        /* Set memory attributes for object       */
          1L,                  /* Two pages (8192 bytes)                 */
          PAG_DEFAULT |        /* Default attributes from allocation     */
          PAG_COMMIT  |        /* Commit page                            */
          PAG_GUARD);          /* Flag page as guard page                */
```

Figure 13. Using a Guard Page With a Memory Object

The example shown in Figure 13 allocates a memory object that is 72KB in size as a sparse object, commits the first two pages (8KB) of the object and specifies the uppermost of the two pages as a guard page. Any attempt by the application to write into this uppermost page will result in a guard page exception.

The guard page exception generated when an application attempts to write into a guard page can be trapped and processed by an application-registered exception handler, to commit further pages within the memory object. A simple guard page exception handler is shown in Figure 14 on page 65.

```
ULONG GPHandler(PEXCEPTIONREPORTRECORD pX)
{
      ULONG ulAttribs;                        /* Memory attributes        */
      ULONG ulSize;                           /* Range in pages           */

      if (pX->ExceptionNum ==                 /* If guard page exception  */
          XCPT_GUARD_PAGE_VIOLATION)
         {
            ulSize=1L;                        /* One page                 */
            DosQueryMem(                      /* Query memory attributes  */
               (PVOID)pX->ExceptionInfo[1],   /* Page base address        */
               &ulSize,                       /* Single page              */
               &ulAttribs);                   /* Memory attributes        */
            if (((ulAttrs & PAG_FREE) ||      /* If page is available     */
                (ulAttrs & PAG_COMMIT))==0)   /* but is not committed     */
               {
               DosSetMem(                     /* Commit page              */
                  (PVOID)pX->ExceptionInfo[1], /* Page base address       */
                  1L,                         /* Single page only         */
                  PAG_DEFAULT |               /* Default attributes       */
                  PAG_COMMIT);                /* Set commit flag          */
               return(XCPT_CONTINUE_EXECUTION); /* Done                    */
               }
         }
      if (pX->ExceptionNum ==                 /* If access violation      */
          XCPT_ACCESS_VIOLATION)
         {
         if (pX->ExceptionInfo[1])            /* If page address not NULL */
            {
            ulSize=1L;                        /* One page                 */
            DosQueryMem(                      /* Query memory attributes  */
               (PVOID)pX->ExceptionInfo[1],   /* Page base address        */
               &ulSize,                       /* Single page              */
               &ulAttribs);                   /* Memory attributes        */
            if (((ulAttrs & PAG_FREE) ||      /* If page is available     */
                (ulAttrs & PAG_COMMIT))==0)   /* but is not committed     */
               {
               DosSetMem(                     /* Commit page              */
                  (PVOID)pX->ExceptionInfo[1], /* Page base address       */
                  1L,                         /* Single page only         */
                  PAG_DEFAULT |               /* Default attributes       */
                  PAG_COMMIT);                /* Set commit flag          */
               return(XCPT_CONTINUE_EXECUTION); /* Done                    */
               }
            }
         }
      return(XCPT_CONTINUE_SEARCH);           /* Chain to next handler if */
}                                             /* any other exception      */
```

Figure 14. Guard Page Exception Handler. This exception handler also handles the situation where an application writes directly to an uncommitted page rather than to the guard page, as is possible with non-sequential write operations.

The exception handler shown in Figure 14 handles two types of exception: the guard page exception and the page fault exception. The latter occurs when an application attempts to write to an uncommitted page in the memory object that is *not* the guard page. This can occur when a memory object is accessed in a non-sequential manner.

The example shown above handles the guard page exception simply by committing the next page in the memory object, and making this page the new guard page. Guard page exceptions should *not* be allowed to pass through to the operating system's default guard page exception handler, since the default handler operates by committing the next *lower* page in the memory object and making this the new guard page. This is done because the default handler is intended mainly to handle dynamic stack growth; stacks are always propagated downward.

The exception handler shown in Figure 14 also handles the page fault exception, where a write operation is attempted into a page other than the guard page, which has not previously been committed. The exception handler responds to this exception by querying the properties of the page in question and, if the page has been allocated but not yet committed, proceeds to commit the page.

If the page has not been allocated (that is, it does not lie within the boundaries of the memory object), or if the exception is neither a guard page exception nor a page fault exception, the exception handler does not process the exception, and returns control to the operating system, which will invoke any other exception handlers registered for the current thread (see 5.4, "Exception Handling" on page 68).

A guard page exception handler is registered by the application using the **DosSetExceptionHandler()** function. This function is illustrated in Figure 15.

```
EXCEPTIONREGISTRATIONRECORD Exception;
   :
   :
Exception.ExceptionHandler = (_ERR *)&GPHandler; /* Set entry point addr */

DosSetExceptionHandler(&Exception);              /* Register handler     */
```

Figure 15. Registering a Guard Page Exception Handler. This example shows the use of the DosSetExceptionHandler() function.

The **DosSetExceptionHandler()** function can also be used to register exception handlers for other types of system exception; see 5.4, "Exception Handling" on page 68 for further information.

Note that OS/2 Version 2.0 provides its own exception handlers within the service layers for all 32-bit system functions. These exception handlers allow the service routines to recover from page fault exceptions and general protection exceptions encountered due to bad pointers in applications' function calls. The function call returns an ERROR_BAD_PARAMETER code rather than a Trap 00D or Trap 00E code, thereby allowing the application to recover. This represents a significant enhancement over previous versions of OS/2, since it allows easier debugging and more flexible pointer handling.

5.3 Suballocating Memory

Under OS/2 Version 2.0, the granular unit of memory is the page. This means that the minimum possible memory allocation for a single **DosAllocMem()** function call is 4KB. For example, if an application requests the allocation of 10 bytes of storage, the operating system will allocate a full 4KB page; the remaining storage in this page will be wasted.

It is therefore recommended that for dynamic allocation of small memory objects for uses such as instance data, each window procedure should use a single **DosAllocMem()** function call to allocate a storage pool, and subdivide this storage as required using the **DosSubAlloc()** function, as shown in Figure 16 on page 67.

```
#define       POOLSIZE   8192            /* Size of storage pool       */
PVOID         pPool;                     /* Base address of pool       */

CTRLSTRUCT1   *Struct1;                  /* Control structure 1        */
CTRLSTRUCT2   *Struct2;                  /* Control structure 2        */

DosAllocMem(&pPool,                      /* Allocate storage for pool  */
            POOLSIZE,                    /* Size of memory object      */
            PAG_READ  |                  /* Allow read access          */
            PAG_WRITE |                  /* Allow write access         */
            PAG_COMMIT);                 /* Commit storage immediately */

DosSubSet(pPool,                         /* Initialize for suballoc    */
          DOS_SUBINIT,                   /* Initialize flag            */
          POOLSIZE);                     /* Size of pool               */

DosSubAlloc(pPool,                       /* Suballocate storage        */
            &Struct1,                    /* Pointer to memory object   */
            sizeof(CTRLSTRUCT1));        /* Size of storage required   */
DosSubAlloc(pPool,                       /* Suballocate storage        */
            &Struct2,                    /* Pointer to memory object   */
            sizeof(CTRLSTRUCT2));        /* Size of storage required   */
```

Figure 16. Suballocating Memory

Storage must be suballocated in multiples of 8 bytes. Any requested suballocation which is not a multiple of 8 bytes will have its size rounded *up* to a multiple of 8 bytes.

Storage to be suballocated must first be allocated using the **DosAllocMem()** function, and initialized for suballocation using the **DosSubSet()** function. Note that control information for the suballocation uses 64 bytes of the storage pool; this must be taken into account when determining the size requirements for the pool.

In Figure 16, the storage in the pool is committed during allocation, since the example assumes that the total storage requirement is known in advance. In situations where the exact size of the storage required is not known, the storage may be allocated but not committed, and the suballocation procedure will then progressively commit storage as required. This is indicated by specifying the DOS_SPARSE_OBJ flag in the **DosSubSet()** function call.

Memory that has been suballocated using the **DosSubAlloc()** function may be freed using the **DosSubFree()** function. The storage is then available for future suballocation. Note, however, that the suballocation procedure does not reorganize suballocated memory objects within a pool. Thus freeing objects within the pool may result in memory fragmentation.

A storage pool initialized for suballocation using the **DosSubSet()** function should be removed using the **DosSubUnset()** function before the memory in the pool is

freed. This function call releases the operating system resources used by the suballocation procedure.

When using the C Set/2 compiler, the **malloc()** function may be used to allocate memory. This function has many of the advantages of the **DosSubAlloc()** function, but avoids the need for the application to explicitly allocate, set and suballocate memory. The **malloc()** function also provides greater independence for application code from the platform upon which it executes, allowing the application to be more easily migrated to platforms other than OS/2 Version 2.0.

The **malloc()** function works as follows:

- The first call to **malloc()** from a particular application (process) causes **malloc()** to request a memory object from the operating system. The **malloc()** service routine adds 16 bytes to the size specified in the function call, and rounds the result upward to the next even power of 2. This amount of memory is then requested from the operating system using a **DosAllocMem()** call. The operating system will then allocate memory, rounding the service routine's request size upward to the nearest multiple of 4KB. The **malloc()** function then fulfills the application's request, with some wastage due to the page-granular allocation.
- For subsequent calls to **malloc()**, the **malloc()** service routine first checks whether it has sufficient memory remaining from a previous request; if so, it allocates that memory and returns control to the application. If not, the service routine requests additional memory from the operating system using the **DosAllocMem()** function.

Note that the **free()** function, used to free memory which has been allocated using **malloc()**, does not return the memory to the operating system; rather, that memory is held by **malloc()** for future use. In order to return memory to the operating system, the application must issue a **heapmin()** function call.

5.4 Exception Handling

The following outcomes are possible when a memory object is referenced by the application:

- If the memory has not been allocated or committed, a general protection exception (Trap 000D) will occur.
- If the memory has been allocated but not committed, a page fault exception (Trap 000E) will occur.

In both of the above cases, the exception is reported to the application's general protection exception handler, if one has been registered by the application. The application may then deal with the error. If an exception handler has not been registered by the application, the default exception handler provided by the operating system will terminate the application.

- If the page to be referenced has been defined as a guard page, a guard page exception is generated. If the application has not registered its own handler for this exception, the system's default handler will commit the page, and mark the next page in the memory object as the new guard page for the object. Once the guard page exception has been processed, execution proceeds normally.
- If none of the above conditions occur, the memory object is accessed and execution proceeds normally.

Exception handlers for the various types of exception may be registered using the **DosSetExceptionHandler()** function, as shown in Figure 15 on page 66.

Note that unlike previous versions of OS/2, application handlers need not be written in assembly language; high-level programming languages may be used.

Exception handlers are registered on a per-thread basis, and multiple exception handlers may be registered for each thread. When more than one exception handler is registered, the handlers are chained, with the most recent addition being placed at the start of the chain. When an exception occurs, control is passed to the first handler, which may handle the exception and return XCPT_CONTINUE_EXECUTION, in which case the operating system returns control to the application.

If the exception handler cannot handle a particular exception, it returns XCPT_CONTINUE_SEARCH, in which case the operating system passes control to the next exception handler in the chain. In this way, control is eventually passed to the operating system's default exception handlers.

When an exception handler is no longer required, it can be removed from the chain using the **DosUnsetExceptionHandler()** function.

Exception handling and the various operating system exceptions that can occur are described in the *IBM OS/2 Version 2.0 Control Program Reference*.

5.5 Shared Memory Objects

By default, memory objects allocated by an application are private to the process in which that application executes. However, OS/2 allows memory to be shared among applications for interprocess communication. Shared memory objects are allocated in a similar manner to private memory objects, using the **DosAllocSharedMem()** function.

Note that while private memory objects are allocated using addresses upward from the lower limit of the process address space, shared memory objects are allocated downward from the upper limit of the process address space. Hence the private and shared memory arenas grow toward one another as more memory objects are allocated during execution.

Shared memory objects may be freed in the same manner as private memory objects, using the **DosFreeMem()** function.

5.5.1 Named versus Anonymous Shared Memory Objects

Shared memory objects may be named or anonymous. Named shared memory objects have names of the form:

```
\SHAREMEM\<objectname.ext>
```

A named shared memory object may be accessed by another process using the **DosGetNamedSharedMem()** function.

An anonymous shared memory object must be declared as "giveable" or "gettable" when it is allocated, in order that it may be made available to other processes using the **DosGiveSharedMem()** or **DosGetSharedMem()** functions. An example is given in Figure 17 on page 70.

```
MYSTRUCT *MYSTRUCT;
APIRET   rc;

rc = DosAllocSharedMem(&MyStruct,        /* Allocate memory object      */
                       NULL,             /* Anonymous memory object     */
                       sizeof(MYSTRUCT), /* Size of memory object       */
                       OBJ_GIVEABLE |    /* Object is giveable          */
                       PAG_WRITE    |    /* Write access is allowed     */
                       PAG_READ     |    /* Read access is allowed      */
                       PAG_COMMIT);      /* Commit storage immediately  */

rc = DosGiveSharedMem(MyStruct,          /* Give access to object       */
                      pidOther,          /* Process to receive access   */
                      PAG_WRITE |        /* Write access is allowed     */
                      PAG_READ);         /* Read access is allowed      */
```

Figure 17. Allocating Shared Memory. This example shows the use of the **DosAllocSharedMem()** *function, declaring a memory object as "giveable."*

The **DosGiveSharedMem()** function can be used at any time to provide another process with a specified level of access to a memory object, provided that the owner of the memory object knows the process ID of the process to which access is to be given.

5.5.2 Committing Shared Memory Objects

Like private memory objects, shared memory objects have a distinction between allocating and committing storage. Shared memory objects may be committed upon allocation, or subsequently using exception handlers and the **DosSetMem()** function. The guard page technique may be used with shared memory objects as well as private memory objects.

One distinction between shared memory objects and private memory objects is that private memory objects may be "de-committed" if the required amount of memory reduces during execution; that is, physical storage is released without releasing the corresponding address ranges in the process address space. Shared memory objects may not be de-committed, to avoid the situation where one process may de-commit a page that is being accessed by another process.

5.6 Summary

Dynamic memory allocation is greatly simplified under OS/2 Version 2.0, since the application developer is no longer required to explicitly code for the 80286 segmented memory model, with its size limitation of 64KB per segment. Larger units of memory may be allocated and manipulated as single units, simplifying application code and reducing development time for applications that manipulate large data structures.

When executable modules compiled for different environments are executed within the same process, the operating system handles interaction between these modules through thunk layers. The conversions made within the thunk layers are transparent to the application modules themselves, and do not require consideration by the application developer. This enables executable files, dynamic link libraries, and resources from different environments to be mixed within the same application.

In general, application developers using OS/2 Version 2.0 are provided with a greater level of function and, at the same time, may take advantage of greatly simplified application development through use of the 32-bit flat memory model, which removes much of the inherent complexity of memory manipulation within the application. Developers may produce applications more efficiently under Version 2.0, and may easily migrate their applications to and from the OS/2 Version 2.0 environment.

Chapter 6. Building a Presentation Manager Application

While the steps necessary to create a Presentation Manager application are generally similar to those required to create any kind of application in the programmable workstation environment under OS/2, there are some specific considerations to be borne in mind with regard to the implementation of object-oriented concepts in Presentation Manager applications, since the Presentation Manager environment does not force the application developer to obey such guidelines. Therefore, this chapter will discuss the implementation of the general concepts outlined in Chapter 4, "The Presentation Manager Application Model," in such a way that they conform to object-oriented principles and achieve the highest level of modularity.

For the purposes of discussion, this chapter will assume that the source code is written using the "C" language. Other programming languages may be used to create Presentation Manager applications while preserving the overall application architecture, provided these languages support the creation of reentrant code and allow recursion.

6.1 Language Considerations

Presentation Manager applications may be written using the following programming languages:

- Assembler language
- "C"
- COBOL/2 (after May 7th 1991)
- FORTRAN (OS/2 Version 1.2 and above)

The use of Assembler language should be avoided wherever possible. While coding to such a low-level language may provide significant performance improvements in critical applications, it is typically more costly in terms of programmer productivity and subsequent code maintenance. Assembler code is also less portable than that written using higher-level languages.

The requirements of the Presentation Manager execution environment restrict the use of some COBOL and FORTRAN compilers. Presentation Manager requires window procedures to be reentrant, and a FORTRAN or COBOL compiler that supports the creation of reentrant code must be used. In addition, much of the default message processing provided by Presentation Manager results in synchronous messages being sent to window procedures. This practice is effectively a recursive subroutine call, and requires window procedures to be written in a language that supports recursion.

In order to create COBOL or FORTRAN source code that executes in the Presentation Manager environment, from a compiler that does not support reentrant or recursive procedures, the application developer must adopt one of two solutions:

1. Create a "C" program to provide the Presentation Manager windowing and dialog management functions, and combine this program with called COBOL or FORTRAN subprograms to perform the actual processing for the application.

2. Create a "winproc-less" application, where a main routine written in COBOL or FORTRAN creates a message-processing loop, captures and explicitly processes all message classes. Such an application has no window procedures.
3. Use the "language support window procedure" provided with the *OS/2 Programmer's Toolkit* under OS/2 Version 1.3, which provides processing for most message classes and returns selected messages to the application for processing.

Where the use of COBOL or FORTRAN is unavoidable, solution (1) above is recommended, since it provides additional flexibility, maintains SAA conformance, retains much of the object-oriented nature of the application, and allows the best use to be made of existing host COBOL or FORTRAN application code, since the subprograms used are invoked using standard language conventions, and data is passed to them using a normal parameter list and returned the same way. The subprograms therefore interact with the calling application in much the same way as an ISPF dialog, minimizing the requirement for modification of existing code and reducing the need to retrain application developers.

Object-oriented programming languages such as Smalltalk and C++ are becoming increasingly popular for the creation of object-oriented code, and are well-suited to the Presentation Manager application model. Organizations may wish to investigate the viability of these languages for particular development projects and environments.

6.2 Function and Data Types

Presentation Manager provides a number of specialized function and data type definitions (such as MRESULT, MPARAM, etc.) for use by Presentation Manager applications. While these type definitions are not "standard" C language types, their use is *strongly* recommended. OS/2 maps these type definitions into standard C language types using *#define* statements embedded in the OS/2 header file *os2.h*. Since the mapping may vary between OS/2 Version 1.3 and Version 2.0 due to differences between the 16-bit and 32-bit operating system architectures, the use of Presentation Manager's type definitions insulates the application source code from the underlying architecture.

6.3 Object-Oriented Programming Practices

While Presentation Manager allows an application developer to implement the fundamental concepts of object-oriented programming in his or her applications, it does not restrict the application developer to the use of these conventions. Therefore to ensure the correct implementation of object-oriented conventions and to enable the maximum level of granularity, a number of guidelines are offered:

- The use of multi-purpose application objects (window procedures) should be avoided; for example, a single window procedure should not handle both user interaction and file access. Manipulation of separate data objects should be achieved using separate window procedures. Background data objects (that is, files or databases) should be manipulated using object windows.

- As a corollary of the above rule, multiple window procedures should not be created to act upon a single data object; where possible, all actions on a particular data object should be performed by a single window procedure. This behavior simplifies any future maintenance should the definition of the logical data entity or its representation change. Note that this guideline may need to be overridden in circumstances where an action requires lengthy processing, in order to preserve application responsiveness.
- The definition, creation and/or establishment of access to data objects should be achieved, where possible, from within a window procedure in order to preserve the concept of data encapsulation. That is to say, the use of global data should be minimized in order to enhance modularity and maximize object independence.
- The input, output and behavior associated with a window procedure should depend solely on the class and contents of the messages it receives, and should not depend on any other external data or parameter, other than a data structure to which a pointer is passed as a message parameter. This preserves the concept of object polymorphism and enhances the potential for reuse.

These guidelines, when obeyed, will enable an application to conform to the established guidelines for object-oriented programming as discussed in Chapter 3, "Object-Oriented Applications."

6.4 Application Main Routine

A sample application main routine is illustrated in Figure 18 on page 76 and Figure 19 on page 77. The functions performed by the main routine are as follows:

1. Register the application to Presentation Manager, and obtain an anchor block handle (that is, an application handle), using the **WinInitialize()** function.
2. Create a message queue, into which Presentation Manager will place all messages intended for the application, using the **WinCreateMsgQueue()** function and passing both the anchor block handle and the required queue size to Presentation Manager, which returns a message queue handle to the application. Note that if the queue size specified is zero (as shown in the example above) then the default queue size of 10 messages is used.
3. Register one or more window classes, for the windows that will be created by the application, and associate a window procedure with each window class, using using the **WinRegisterClass()** function. Parameters passed to this function include the name of the window class and the name of the window procedure to be associated with the class. Presentation Manager returns a Boolean value indicating success or failure. Note the 4 bytes (32 bits) requested for window words, which may be used by the window procedure to store the address of its instance data block.

```
#define  INCL_WIN
#include <os2.h>

#define  WCP_MAIN     "WCP_MAIN"

MRESULT EXPENTRY wpMain(HWND,ULONG,MPARAM,MPARAM);

int main()
{
    struct MYSTRUCT InitData;

    static CHAR szTitle[] = "Main Window";

    FRAMECDATA fcdata;                  /* Control data for window   */
    HAB    hAB;                         /* Anchor block handle       */
    HMQ    hMsgQ;                       /* Message queue handle      */
    HWND   hFrame, hClient;             /* Window handles            */
    QMSG   qMsg;                        /* Message queue structure   */
    APIRET rc;                          /* Flag                      */

    memset(&fcdata,0,sizeof( fcdata );  /* Initialize                */
    fcdata.cb      = sizeof( fcdata );

    hAB   = WinInitialize(0);           /* Register appl. to PM      */
    hMsgQ = WinCreateMsgQueue(hAB,0);   /* Create message queue      */

    rc = WinRegisterClass(hAB,          /* Register window class     */
                          WCP_MAIN,     /* Name of class             */
                          wpMain,       /* Window procedure name     */
                          0L,           /* No style                  */
                          4);           /* 32 bits in window words   */
```

Figure 18. Sample Application Main Routine (Part 1) - Registration

4. Create a main display window for the application, using two consecutive **WinCreateWindow()** calls (as shown in Figure 19 on page 77) or a single **WinCreateStdWindow()** call. Note the separate handles used for the frame and client windows. The values specified for *fcdata.flCreateFlags* control the appearance of the window, the controls it contains and its position on the screen.

5. Optionally, create an entry for the application in the Workplace Shell Window List, using the **WinAddSwitchEntry()** function. Note that this step is omitted from Figure 19 for reasons of clarity, and is shown separately in Figure 20 on page 78.

   Note that under OS/2 Version 2.0, the **WinCreateSwitchEntry()** function is provided in addition to the **WinAddSwitchEntry()** function. These two functions accept identical parameters and carry out identical tasks; the **WinCreateSwitchEntry()** function is intended to provide consistent function naming conventions. The **WinAddSwitchEntry()** function is retained under OS/2 Version 2.0 for compatability with existing applications, but use of the **WinCreateSwitchEntry()** function is recommended.

6. Establish a message processing loop, whereby the application requests Presentation Manager to supply messages from the system queue and subsequently invokes Presentation Manager to dispatch them to the

appropriate window procedure. This loop uses nested **WinGetMsg()** and **WinDispatchMsg()** calls.

7. Upon receivng the special message class WM_QUIT, which will cause **WinGetMsg()** to return false and hence terminate the *while* loop, remove any remaining windows using the **WinDestroyWindow()** function, remove the application's entry from the Window List using the **WinRemoveSwitchEntry()** function, destroy the message queue and deregister the application before terminating. These latter functions are achieved using the **WinDestroyMsgQueue()** and **WinTerminate()** calls.

```
    fcdata.flCreateFlags = FCF_TITLEBAR      |   FCF_SYSMENU   |
                           FCF_SIZEBORDER    |   FCF_MINMAX    |
                           FCF_SHELLPOSITION;

    hFrame=WinCreateWindow(HWND_DESKTOP,    /* Create frame window       */
                           WC_FRAME,        /* Frame window class        */
                           (PSZ)0,          /* No window text            */
                           0L,              /* No style                  */
                           0,0,0,0,         /* PM shell will position    */
                           (HWND)0,         /* No owner                  */
                           HWND_TOP,        /* On top of siblings        */
                           0,               /* No window identifier      */
                           &fcdata,         /* Frame control data        */
                           0);              /* Presentation parameters   */
    hClient=WinCreateWindow(hFrame,         /* Create client window      */
                            WCP_MAIN,       /* Window class              */
                            szTitle,        /* Window title              */
                            01,             /* Standard style            */
                            0,0,0,0,        /* PM shell will position    */
                            (HWND)0,        /* No owner                  */
                            HWND_TOP,       /* On top of siblings        */
                            FID_CLIENT,     /* Client window identifier  */
                            &InitData,      /* Initialization data       */
                            0);             /* Presentation parameters   */

    /*    <Create Window List entry for application>                     */

    while (WinGetMsg(hAB, &qMsg, 0, 0, 0))
          WinDispatchMsg(hAB, &qMsg);

    /*    <Remove Window List entry for application>                     */

    WinDestroyWindow(hFrame);               /* Destroy frame & children  */
    WinDestroyMsgQueue(hMsgQ);              /* Destroy message queue     */
    WinTerminate(hAB);                      /* Deregister application    */
}
```

Figure 19. Sample Application Main Routine (Part 2) - Window Creation

The structure of the main routine is similar for both the application (that is, the application's primary thread) and any secondary threads created by the application. See Chapter 10, "Multitasking Considerations" for further discussion on secondary threads.

In Figure 18, note the use of the EXPENTRY keyword in the function prototype to specify the *system* linkage convention for the window procedure *wpMain*. This is required whenever declaring a window procedure or dialog procedure, since

such procedures are normally invoked by Presentation Manager on the application's behalf, rather than directly by the application.

If the application is to appear in and be selectable from the Workplace Shell Window List, the main routine must issue a **WinAddSwitchEntry()** function call, after creating the application's main window and before entering the message processing loop.[3] This function call is shown in Figure 20.

```
SWCNTRL   SwitchData;                          /* Switch control data block  */
HSWITCH  hSwitch;                              /* Switch entry handle        */
   :
   :
SwitchData.hwnd = hFrame;                      /* Set frame window handle    */
SwitchData.hwndIcon = 0;                       /* Use default icon           */
SwitchData.hprog = 0;                          /* Use default program handle */
SwitchData.idProcess = 0;                      /* Use current process id     */
SwitchData.idSession = 0;                      /* Use current session id     */
SwitchData.uchVisibility = SWL_VISIBLE;        /* Make visible               */
SwitchData.fbJump = SWL_JUMPABLE;              /* Make jumpable via Alt+Esc  */
SwitchData.szSwTitle[0] = '\0';                /* Use default title text     */

hSwitch = WinAddSwitchEntry(&SwitchData);      /* Add switch entry           */
```

Figure 20. WinAddSwitchEntry() Function. This function adds the application to the OS/2 Window List. Note that under OS/2 Version 2.0, the WinCreateSwitchEntry() function should be used.

Note that the application may set the *swTitle* field of the *SwitchData* structure to NULL. Presentation Manager will then determine the title under which the application was started from the Presentation Manager shell, and use this title for the switch entry.

The **WinAddSwitchEntry()** function returns a switch entry handle, which may be stored by the application and used during termination to remove the switch entry from the Workplace Shell Window List using the **WinRemoveSwitchEntry()** function.

The switch entry may be accessed by a window procedure at any time during application execution. The switch entry handle is obtained using the **WinQuerySwitchHandle()** function, and the *SwitchData* control structure may then be obtained using the **WinQuerySwitchEntry()** function, and altered using the **WinChangeSwitchEntry()** function. This capability may be used to allow a window procedure to obtain the handle of the application's main window, in order to post or send messages to that window. This is discussed in 6.6.5, "Identifying the Destination Window" on page 91.

[3] Note that under OS/2 Version 2.0, use of the **WinCreateSwitchEntry()** function is recommended, for reasons of consistency in function names.

6.5 Using Windows

As mentioned in 4.3.2, "Window Procedures" on page 46, window procedures within a Presentation Manager application are reentrant; that is, the same window procedure is used for multiple instances of the same window class. However, a window class may have separate data objects associated with each instance of that class, which may be used to store temporary data necessary during the existence of that object; such data is known as instance data. These data objects may need to be created/opened and initialized. Upon the window being closed, data objects may need to be closed or destroyed in a controlled fashion.

Presentation Manager allows such function to be performed by a window procedure, since messages are sent to a window by Presentation Manager informing the window procedure of events such as creation or closure of the window. These messages are discussed below.

6.5.1 Window Creation

A window is created by Presentation Manager in response to the application issuing a **WinCreateStdWindow()** function call or a **WinCreateWindow()** call; an example of the **WinCreateWindow()** call is given in Figure 19 on page 77.

The first statement in the example specifies the attributes of the frame window, which are contained in the data variable *fcdata.flCreateFlags*. These values determine the control windows that are created with the frame window (FCF_SYSMENU, FCF_MINMAX etc), and also indicate to Presentation Manager that it should select the position of the window on the desktop (FCF_SHELLPOSITION).

The window is then created in two steps; firstly the frame window is created, with the desktop as its parent, and then the client window is created with the frame window as its parent. The frame window belongs to the system-defined window class WC_FRAME, whereas the client window belongs to an application-defined window class WCP_MAIN, which is assumed to have already been defined to Presentation Manager using a **WinRegisterClass()** call.

If it is necessary to pass initialization information to a window upon its creation, this may be achieved using the *CtlData* parameter in the **WinCreateWindow()** function. This parameter is a 32-bit pointer, which may reference an application-defined data structure. This pointer is passed to the window as the first parameter of the WM_CREATE message. The window may, during its processing of this message, extract the pointer from the message parameter and use it to access the data structure. See Figure 19 for an example of this technique.

When an application requests that Presentation Manager creates a window of a particular class, a message of the system-defined class WM_CREATE is sent to the window procedure associated with that class. The window procedure may capture this message by including a case for it, and perform any processing such as opening files or databases, allocating memory objects and setting instance data to initial default values.

In coding the method for this message class, the first statement should be a call to **WinDefWindowProc()**, which will enable Presentation Manager to perform default processing and complete the initialization of the window (such as

allocating a window handle) before the application-specific processing is carried out. If the default processing is not completed first, the window handle and any window words may not be allocated before the application makes function calls that reference them, thus causing these calls to fail.

Where instance data or resource handles will be used by the window, and must be maintained beyond the processing of a single message, a data structure should be defined to contain these items. Memory for the data structure should be requested from the operating system, and a pointer to the memory object stored in the window words, as part of the WM_CREATE processing. See 6.5.4, "Instance Data and Window Words" on page 81 for further information.

6.5.2 Window Processing

During execution, a window processes messages passed to it by Presentation Manager, using the methods defined in its window procedure. Upon receiving a message, the window procedure performs three basic tasks:

1. The window procedure determines the message class by examining the message class identifier.
2. Depending upon the message class, the window procedure executes a series of application instructions and/or subroutines to perform the action requested by the message.
3. The window procedure passes a return code to Presentation Manager.

As part of the second step above, the window procedure may extract necessary information from the parameters passed with the message, using a number of macros provided by Presentation Manager. These macros are described in 6.6.6, "Creating Message Parameters" on page 93.

The window procedure may also gain access to instance data or resource handles stored in a control block during processing of previous messages. This control block is generally allocated upon creation of the window and a pointer to it stored in the window words. The window procedure may retrieve this pointer from the window words at the start of processing for the current message. See6.5.4, "Instance Data and Window Words" on page 81.

6.5.3 Window Closure

A window is closed (removed from the screen and destroyed) by Presentation Manager in response to the application issuing a **WinDestroyWindow()** call, specifying the handle of the window to be destroyed. In normal circumstances the handle of the frame window is specified; destroying the frame window destroys that window and all of its children, including the client window associated with that frame.

When an application requests that Presentation Manager close a window, a system-defined message of class WM_DESTROY is sent to the client window, and thus to the window procedure associated with that class. The window procedure may capture and process this message, backing out any uncompleted units of work, and destroying or terminating access to data objects. The window procedure should then return a value of zero.

Note that although closing and destroying a parent window will also close and destroy all children of that window, the WM_DESTROY message is sent to the parent window, and processed before the children are destroyed. Hence when

processing a WM_DESTROY message, a window procedure may assume that all its children still exist.

If the user explicitly requests closure of a window by selecting the "Close" option on the system menu, a system-defined message of class WM_CLOSE is sent to the window procedure. The window procedure may also capture and process this message in a similar manner to that used for WM_DESTROY messages.

Note that explicit processing of the WM_CLOSE message class is recommended for *all* Presentation Manager windows, since the default processing provided by Presentation Manager causes a WM_QUIT message to be posted to the application's message queue. This may result in unwarranted termination of the application. The window procedure for a child window should process a WM_CLOSE message by issuing a **WinDestroyWindow()** call for its frame window. The window procedure for an application's main window should process a WM_CLOSE message by posting a WM_QUIT message to itself. This will cause the application to terminate (see 6.8, "Terminating an Application" on page 98).

In order to handle the closure of a window in the most elegant manner, the following course of action is recommended:

- Explicit processing should be provided for both WM_CLOSE and WM_DESTROY messages:
  - A window procedure should process a WM_CLOSE message by issuing a **WinDestroyWindow()** call for its own frame window if it is a child window, or a WM_QUIT message to itself if it is an application's main window. In both cases, the window procedure should then return a value of zero.
  - A window procedure should process a WM_DESTROY message by closing any files or databases that it has opened, and freeing any resources such as memory objects.
- Selection of the "Exit" option from a menu bar should result in the closure of the window to which the menu bar belongs, by having the window procedure issue a **WinDestroyWindow()** call for its frame window. If the window is the application's main window, it should be closed by having the window procedure post a WM_QUIT message to itself (see 6.8, "Terminating an Application" on page 98). This will result in a WM_DESTROY message being posted to the main window and each of its children as part of the application's termination processing. These messages may be captured and processed by the appropriate window procedures in order to close data objects, back out incomplete units of work, etc.

The release of data objects and Presentation Manager resources is discussed in 6.5.4, "Instance Data and Window Words."

6.5.4 Instance Data and Window Words

For data that is private to a particular instance of a window class, each window may have an area of storage associated with it, assigned by Presentation Manager and located within the Presentation Manager control block for that window. This area is known as the **window words**. The amount of space allocated for window words in a particular window class is variable, and is defined in the **WinRegisterClass()** function call at the time the class is registered to Presentation Manager.

It is recommended that for storage of amounts of data larger than four bytes, a memory object is obtained from the operating system using the **DosAllocMem()** or **DosSubAlloc()** functions, and a pointer to this object is placed in the window ' words of the associated window. An example of this technique is given in Figure 21 on page 82.

```
MYSTRUCT *MyStruct;
   :
switch (ulMsg)                              /* Switch on message class   */
       {
       case WM_CREATE:
            WinDefWindowProc(hWnd,          /* Perform default init      */
                             ulMsg,
                             mp1,
                             mp2);
            DosAllocMem(MyStruct,           /* Allocate memory object    */
                        sizeof(MYSTRUCT),   /* Size of memory object     */
                        PAG_READ  |         /* Allow read access         */
                        PAG_WRITE |         /* Allow write access        */
                        PAG_COMMIT);        /* Commit storage now        */
            hFrame=WinQueryWindow(hwnd,     /* Get frame window handle   */
                                  QW_PARENT,
                                  FALSE);
            WinSetWindowULong(hFrame,       /* Place pointer in window   */
                              QWL_USER,     /* words                     */
                              (ULONG)MyStruct);
            return((MRESULT)0);
            break;
              :
```

Figure 21. Storing Instance Data in Window Words. This example shows the allocation of a memory object, and the storage of a pointer to that memory object in window words.

A memory object corresponding to the size of the data structure MYSTRUCT is obtained from the operating system using the **DosAllocMem()** function, and a pointer to this memory object is set by the application. This pointer is then placed in the window words of the current window's parent (that is, the frame window) using the **WinSetWindowULong()** function, at offset QWL_USER. A number of predefined Presentation Manager window classes, including the frame window class, contain a 32-bit word at this offset, which is available for application use.

Note the use of the PAG_COMMIT flag in the **DosAllocMem()** function call. This flag causes storage to be allocated immediately for the memory object being created, since OS/2 Version 2.0 by default uses a two-phase process for dynamic memory allocation.

The concept of committing memory is new to Version 2.0, and allows a storage map for the application to be defined, but the storage is not reserved in memory until it is needed, at which time the application may explicitly commit the storage using the **DosSetMem()** function. Optionally, the application may set the PAG_COMMIT flag in the **DosAllocMem()** function call to commit the storage immediately upon allocation.

Failure to commit storage, either by use of the PAG_COMMIT flag or the **DosSetMem()** function, will result in a page fault exception (Trap 000E) when the application attempts to write to the storage area. The concept of allocating and

committing storage is explained fully in *OS/2 Version 2.0 - Volume 1: Control Program*, and the use of these techniques by applications is described in Chapter 5, "The Flat Memory Model."

After the memory object containing instance data is initially allocated, the window procedure may access it during processing of subsequent messages by issuing a **WinQueryWindowULong()** call to Presentation Manager, as shown in Figure 22.

```
case WMP_MYMESSAGE:
     hFrame=WinQueryWindow(hwnd,
                           QW_PARENT,
                           FALSE);
     MyStruct=WinQueryWindowULong(hFrame,
                                  QWL_USER);
     <Perform action>
     return((MRESULT)0);
     break;
       :
```

Figure 22. Retrieving Instance Data from Window Words

Upon termination of the window by the application, the window procedure receives a WM_DESTROY message. As described in 6.5.3, "Window Closure" on page 80 , the window procedure should process this message by releasing any resources to which it has access. This includes the instance data control block, which must be released using the **DosFreeMem()** function as shown in Figure 23.

```
case WM_DESTROY:
     hFrame=WinQueryWindow(hwnd,
                           QW_PARENT,
                           FALSE);
     MyStruct=WinQueryWindowULong(hFrame,
                                  QWL_USER);
     <Release data objects>
     <Release Presentation Manager resources>
     DosFreeMem(MyStruct);
     return((MRESULT)0);
     break;
       :
```

Figure 23. Releasing Instance Data Storage

In the above example, the pointer to the instance data control block is first retrieved from the window words, giving access to the handles of any data objects or Presentation Manager resources obtained by the window, in order that these may be released. Once this has been achieved, the memory object containing the control block is released by the window procedure. Failure to release the data objects and resources before freeing the memory object would result in a general protection exception (Trap 000D) when the data objects or resources were subsequently released.

6.5.5 Subclassing a Window

The use of subclassing to modify the methods of an existing window class has been described in 4.7, "Subclassing" on page 57. An application subclasses a particular window instance (rather than the entire window class) by creating a **subclass window procedure**, and registering this window procedure to Presentation Manager using the **WinSubclassWindow()** function.

The use of the **WinSubclassWindow()** function is shown in Figure 24.

```
PFNWP pOldWinProc;

pOldWinProc = WinSubclassWindow(hWnd, wpSubclass);
```

Figure 24. WinSubclassWindow() Function

The **WinSubclassWindow()** function substitutes a new window procedure, known as the **subclass window procedure**, for the original window procedure associated with the window being subclassed. The window handle of the window, along with the entry point of the subclass window procedure, is passed to the **WinSubclassWindow()** function. The function returns the entry point address of the original window procedure for that window.

Once a window has been subclassed, Presentation Manager routes messages destined for that window to the subclass window procedure. The subclass window procedure may:

- Process the message itself, if it indicates an action for which the method must be modified.

  The subclass window procedure then returns control immediately to Presentation Manager.

- Pass the message on to the original window procedure for that window, if the subclass window procedure is not explicitly concerned with the action indicated by the message.

  The original window procedure is directly invoked by the subclass window procedure; note that this is one of the few instances where direct invocation of a window procedure is recommended. The return code from the original window procedure is then returned to Presentation Manager.

- Both of the above, if the subclass window procedure must perform some processing in addition to that normally performed by the original window procedure.

  The additional processing performed by the subclass window procedure may be performed either before or after the processing performed by the original window procedure. This sequence is at the discretion of the application developer, and depends largely on the desired modification in the window's behavior.

A subclass window procedure is similar in structure to a "normal" window procedure, except that instead of calling the **WinDefWindowProc()** function as its default case, it should invoke the original window procedure. This means that the entry point address of the original window procedure must be known to and accessible from the subclass window procedure. Note also that the entry point address might not be that of the original window procedure specified when the window class was registered to Presentation Manager, since the window might

previously have been subclassed, and the current subclassing operation might be effectively subclassing the subclass window procedure.

The entry point address of the original procedure can be supplied to the subclass window procedure in a number of ways:

- It may be determined from the information returned by the **WinSubclassWindow()** call, and passed to the subclass window procedure in an application-defined message. The subclass window procedure may then store the entry point address in a global variable or in the window words of the window, assuming the available window words are not already in use.
- It may be determined by the subclass window procedure itself by querying Presentation Manager. Note, however, that this method will only work if the window has not previously been subclassed, since Presentation Manager only records the original window procedure (as specified in the **WinRegisterClass()** function call) in the CLASSINFO structure for the window.

An example of a subclass window procedure, including a query to obtain the original entry point address from the Presentation Manager class information, is given in Figure 25 on page 86.

```
MRESULT EXPENTRY wpSubclass(HWND   hWnd,
                            ULONG  ulMsg,
                            MPARAM mp1,
                            MPARAM mp2)
{
        CHAR      szClass[7];
        CLASSINFO WinClass;
        PFNWP     pWinProc;

        BOOL      bSuccess;
        ULONG     ulRetLength;

        switch (ulMsg)
               {
               case WMP_MESSAGE1:
                         :
                    <Perform application specific processing>
                         :
                    return((MRESULT)0);
                    break;
               case WMP_MESSAGE2:
                         :
                    <Perform application specific processing>
                         :
                    break;
               default:
                    break;
               }
        ulRetLength=WinQueryClassName(hWnd,
                                      sizeof(szClass),
                                      szClass);
        bSuccess=WinQueryClassInfo(NULL,
                                   szClass,
                                   &WinClass);
        pWinProc=WinClass.pfnWindowProc;
        return((MRESULT)(*pWinProc)(hWnd,
                                    ulMsg,
                                    mp1,
                                    mp2);
}
```

Figure 25. Subclass Window Procedure

Figure 25 shows each of the possible cases listed above. The message class WMP_MESSAGE1 is explicitly processed by the subclass window procedure, which then returns control to Presentation Manager with a *return* statement upon completion.

The message class WMP_MESSAGE2 is also explicitly processed by the subclass window procedure, but in this case it is required that the processing performed by the original window procedure be allowed to occur, after the subclass window procedure's processing. The subclass window procedure therefore does not return control immediately to Presentation Manager, but merely terminates the *switch* statement, allowing the final four statements to be executed.

For other message classes with which the subclass window procedure is not concerned, the default case also terminates the *switch* statement, allowing the final four statements to be executed.

These final statements determine the entry point address of the original window procedure, using the **WinQueryClassName()** and **WinQueryClassInfo()** functions to access control information held by Presentation Manager. This entry point address is then used to invoke the original window procedure to process messages with which the subclass window procedure is not concerned, or for which the normal processing must be allowed to occur.

The last four statements in the example above are common to all subclass window procedures, and organizations undertaking development of Presentation Manager applications may wish to incorporate them into a standard subroutine and place them in a library for access by developers.

Note that a subclass window procedure, like all window and dialog procedures, must use the *system* linkage convention. This is normally achieved by declaring the subclass window procedure using the EXPENTRY keyword.

6.6 Window Communication

Presentation Manager provides a number of mechanisms for communicating between windows. All of these mechanisms use the Presentation Manager message concept. The exact technique used in any particular situation is dependent upon the nature of the communications and the types of windows involved.

6.6.1 Standard Windows

Data may be passed to a window upon its creation, using the *CtlData* parameter of the **WinCreateWindow()** function. The contents of this parameter (a 32-bit pointer) are passed to the target window as a parameter to the WM_CREATE message. The contents may then be extracted from the message parameter and used by the window procedure.

When an application wishes to pass a message between two standard windows that currently exist, whether they are display windows or object windows, either of two methods may be used, depending on whether the desired communication is to be synchronous or asynchronous.

- When a synchronous message is to be passed, the **WinSendMsg()** function is used, and the target window procedure is invoked directly by Presentation Manager, in a similar fashion to a normal function call. The return code from the window procedure is passed by Presentation Manager to the calling window procedure, where it may be interrogated and acted upon.
- When a message is to be processed asynchronously, the **WinPostMsg()** function is used. In this case the message is posted to a queue associated with the thread that created the target window, and the return code to the calling window procedure merely indicates that the message was successfully placed on the queue. In order for the target window procedure to pass a return code or acknowledgement back to the calling window procedure, it must include another **WinPostMsg()** call as part of the processing of the message.

The use of **WinPostMsg()** is recommended over that of **WinSendMsg()**, since posted messages are processed in the order in which they arrive in the queue, and the integrity of the user's intention is thus preserved in the order of processing. In addition, synchronous window procedures are invoked and executed without the original window procedure completing its processing and returning control to the message processing loop. Thus the application is prevented from processing additional user interaction, which may lead to violation of the SAA CUA responsiveness guidelines.

6.6.2 Dialog Boxes

Communication between a standard window and a modeless dialog box is achieved in a similar fashion to that used between two standard windows, since the modeless dialog box is merely a normal window without a sizable border. However, communication between a standard window and a modal dialog box must be achieved in a different manner, since a modal dialog box is typically loaded and processed in a single **WinDlgBox()** function call, and the dialog box only has an existence during the execution of that function call. An example of the **WinDlgBox()** function is shown in Figure 26.

```
MYSTRUCT *MyStruct;
   :
DosAllocMem(MyStruct,                  /* Allocate memory object          */
            sizeof(MYSTRUCT),          /* Size of memory object           */
            PAG_READ  |                /* Allow read access               */
            PAG_WRITE |                /* Allow write access              */
            PAG_COMMIT);               /* Commit storage now              */

<Initialize values in MyStruct>        /* Set initialization data         */

rc = WinDlgBox(HWND_DESKTOP,           /* Desktop is parent               */
               hwnd,                   /* Current window is owner         */
               dpMyDialog,             /* Entry point of dialog procedure */
               (HMODULE)0,             /* Resource is in EXE file         */
               DC_MYDIALOG,            /* Dialog resource identifier      */
               MyStruct);              /* Pointer to initialization data  */
```

Figure 26. WinDlgBox() Function

Data may be passed to a dialog procedure at initialization time by creating a data structure and passing a pointer to that structure in the *CreateParams* field of the **WinDlgBox()** function, as shown in Figure 26. This pointer is passed to the dialog procedure as the second parameter of the WM_INITDLG message, and may be accessed by the dialog procedure during the processing of this message. Note that this is the *only* time at which input may be passed to a dialog box, since the dialog is processed within the scope of a single application statement; either a **WinDlgBox()** call or a **WinProcessDlg()** call may be used. The WM_INITDLG message is described in the *IBM OS/2 Version 2.0 Presentation Manager Reference.*

Information may be conveyed from a dialog procedure to its calling window procedure in one of two ways:

- The dialog box may provide an unsigned integer (USHORT) parameter to the **WinDismissDlg()** function, and this value is passed to the calling window procedure as the return code from the **WinDlgBox()** function. This technique

is useful where an acknowledgement or simple return data must be conveyed.

- The dialog box may issue a **WinPostMsg()** call to pass a message to the queue associated with its calling window. The window procedure may then receive and process that message in the normal way. This technique is useful when more complex data or structures must be conveyed.

The latter technique above may also be used to convey information to a window other than the window that invoked the dialog. This may be necessary in situations where a dialog box is invoked by one window procedure on behalf of a group of windows.

6.6.3 Control Windows

As mentioned in Chapter 11, "Systems Application Architecture CUA Considerations," control windows are typically used in dialog boxes, and are hence accessed from the dialog procedure associated with their parent dialog box. Such communication is synchronous in nature, since it usually involves insertion or retrieval of data into or from control windows, or other tasks that are part of the modal dialog with the user.

Under OS/2 Version 2.0, some additional functions have been introduced into the Presentation Manager programming interface, to ease more complex communications, such as those involving list boxes. Since communication with list boxes is therefore somewhat different from that involving other control window classes, list boxes are discussed separately in 6.6.3.2, "List Boxes" on page 90.

6.6.3.1 General Control Windows

Communication between a dialog procedure and the control windows associated with its dialog box is typically achieved using the **WinSendDlgItemMsg()** function, which is documented in the *IBM OS/2 Version 2.0 Presentation Manager Reference*. This function is similar in function and behavior to the **WinSendMsg()** function, in that it passes a synchronous message to the destination window. However, instead of accepting the handle of the destination window as its first parameter, it accepts the handle of the control window's parent and the window identifier of the control window itself as the first two parameters of the call. For example, to send a message of class EM_SETTEXTLIMIT to an entry field named EF_PRODNAME, which is a child of the dialog box with handle *hDlgBox*, the function call shown in Figure 27 is used:

```
rc = WinSendDlgItemMsg(hDlgBox,                /* Parent dialog box    */
                       EF_PRODNAME,            /* Control identifier   */
                       EM_SETTEXTLIMIT,        /* Message              */
                       20,                     /* Message parameters   */
                       0);
```

Figure 27. Communicating with a Control Window

It is possible to perform an equivalent function using the **WinSendMsg()** call, by obtaining the control window's handle using the **WinWindowFromID()** function. However, for purposes of standardization and in accordance with emerging conventions, it is recommended that the **WinSendDlgItemMsg()** function be used to send messages to control windows. Note that for this purpose, the definition of control windows includes both the system menu and menu bar; messages

sent to these menus (in order to insert, modify or delete items) should be sent using the **WinSendDlgItemMsg()** function.

Similarly, it is recommended that the **WinSetDlgItemText()** and **WinQueryDlgItemText()** functions be used to set and query the contents of control windows from within the application. For example, assume that the user has completed interaction with a dialog box, and pressed the "Enter" or "OK" button, and the application wishes to obtain the contents of an entry field named EF_PRODNAME, which is child of the dialog box with handle *hDlgBox*. The function call call shown in Figure 28 is used.

```
rc = WinQueryDlgItemText(hDlgBox,                /* Parent dialog box  */
                         EF_PRODNAME,            /* Control identifier */
                         sizeof(szBuffer),       /* Size of buffer     */
                         szBuffer);              /* Pointer to buffer  */
```

Figure 28. Querying Information From a Control Window

The **WinQueryDlgItemText()** function copies the contents of the entry field into the string *szBuffer*, and returns the number of characters copied.

The **WinSetDlgItemText()** function is typically used in situations where some of the information necessary to complete an action is known; this information is then displayed in the appropriate entry fields within the dialog box, and the user fills in the missing fields. Another use of this function is to provide default values for entry fields. Both the **WinSetDlgItemText()** and **WinQueryDlgItemText()** functions are documented in the *IBM OS/2 Version 2.0 Presentation Manager Reference*.

6.6.3.2 List Boxes

The complexity of communication with list boxes has been greatly reduced under OS/2 Version 2.0. The Presentation Manager programming interface now includes a number of functions that allow most communication requirements to be achieved in a single step. Note that these functions may also be used for communication with a combo box (prompted entry field).

Insertion and deletion of list box items is carried out using the **WinInsertLboxItem()** and **WinDeleteLboxItem()** functions, which are new to OS/2 Version 2.0. The **WinInsertLboxItem()** function is illustrated in Figure 29.

```
hLBox = WinWindowFromID(hWnd,              /* Get list box window handle */
                        LB_LIST);

ulIndex = WinInsertLboxItem(hLBox,         /* Insert list box item       */
                            LIT_END,       /* Insert at end of list      */
                            szItemText);   /* Item text                  */
```

Figure 29. Inserting an Item Into a List Box

An application may obtain the text of a selected item in the list box using the **WinQueryLboxSelectedItem()** and **WinQueryLboxItemText()** functions. The use of these functions is illustrated in Figure 30 on page 91.

```
hLBox = WinWindowFromID(hWnd,              /* Get list box window handle */
                        LB_LIST);

ulIndex = WinQueryLboxSelectedItem(hLBox); /* Get index of selected item */

ulLength = WinQueryLboxItemText(hLBox,            /* Get item text     */
                                usIndex,          /* Index of item     */
                                szBuffer,         /* Text buffer       */
                                sizeof(szBuffer)); /* Max no. of chars  */
```

Figure 30. Querying a Selected List Box Item

Other functions include the **WinQueryLboxCount()** function, which returns the number of items in a list box, and the **WinQueryLboxItemTextLength()** function, which returns the length of list box item's text.

All of these list box manipulation functions are described in the *IBM OS/2 Version 2.0 Presentation Manager Reference.*

6.6.4 Message Boxes

Communication between a window or dialog procedure and a message box is relatively simple. The message box is created and processed using the **WinMessageBox()** function, and the only input data provided to this function is the title of the message box and the text of the message to be displayed. The application may affect the style of the message box, by specifying style attributes in the function call, as described in the *IBM OS/2 Version 2.0 Presentation Manager Reference.*

An example of the **WinMessageBox()** function is given in Figure 31.

```
rc = WinMessageBox(HWND_DESKTOP,        /* Desktop is parent           */
                   hWnd,                /* Current window is parent    */
                   pszMsgText,          /* Pointer to message text     */
                   "Open the File"      /* Message title               */
                   0,                   /* Message box identifier      */
                   MB_OKCANCEL   |      /* Include OK & Cancel buttons */
                   MB_DEFBUTTON1 |      /* Default to OK               */
                   MB_HELP);            /* Include help button         */
```

Figure 31. WinMessageBox() Function

The result of the user's interaction with the message box (that is, the identifier of the button that was pressed) is communicated to the application in the form of an unsigned integer returned by the **WinMessageBox()** call. The application may then interrogate this returned value to determine the subsequent action to be taken.

6.6.5 Identifying the Destination Window

When passing messages between windows using the **WinPostMsg()** or **WinSendMsg()** functions, the window handle of the destination window must be known and specified in the message. If window handles are not defined globally, the required handle must be obtained from Presentation Manager. This may be achieved in a number of ways:

- If the target window has a known relationship to the current window or to another window for which the handle is already known, the **WinQueryWindow()** function may be used to obtain the window handle of the target window. For example, if a window wishes to post a message to its own parent window, the technique shown in Figure 32 may be used.

```
hTarget = WinQueryWindow(hWnd,        /* Base window for relation  */
                         QW_PARENT,   /* Relationship to base wndw */
                         FALSE);      /* Do not lock window        */
```

Figure 32. Obtaining a Window Handle - WinQueryWindow() Function

  The **WinQueryWindow()** call returns the handle of the required window. Relationships other than parent/child may also be used by this function; the valid relationships are described, along with the **WinQueryWindow()** function, in the *IBM OS/2 Version 2.0 Presentation Manager Reference.*

- If the parent window and window identifier of the target window are known, the **WinWindowFromID()** function may be used to obtain the window handle of the target window. For example, if a window wishes to post a message to the client window of its application's main window, assuming the frame window handle is known, the method shown in Figure 33 may be used.

```
hTarget = WinWindowFromID(hMainFrame,  /* Parent window known     */
                          FID_CLIENT); /* Window identifier       */
```

Figure 33. Obtaining a Window Handle - WinWindowFromID() Function

  The **WinWindowFromID()** function also returns the handle of the required window.

- If the target window is the application's main window, its handle may be obtained by first querying the application's switch entry in the Workplace Shell Window List to obtain the handle of the main frame window (using the **WinQuerySwitchHandle()** and **WinQuerySwitchEntry()** functions), then using the **WinWindowFromID()** function to obtain the handle of the client window, as shown in Figure 34.

```
hSwitch = WinQuerySwitchHandle(hWnd,0);
ulSuccess = WinQuerySwitchEntry(hSwitch,
                                SwitchData);
hTarget = WinWindowFromID(SwitchData.hwnd,
                          FID_CLIENT);
```

Figure 34. Obtaining a Window Handle Using the Switch Entry

  The above example assumes that the application has been added to the OS/2 Window List using the **WinAddSwitchEntry()** function, and the handle of its main frame window supplied as a parameter. See Figure 20 on page 78.

When passing messages synchronously to control windows using the **WinSendDlgItemMsg()** function, it is generally assumed that the target control window is a child of the current window or dialog box. Thus the parent window handle is the handle of the current window, and the window identifier is also known to the current window procedure. An exception is the case where a

window procedure wishes to send a message to a frame control of its own parent frame window. In this case a **WinQueryWindow()** call must be issued with the QW_PARENT parameter to determine the handle of the frame window. The **WinSendDlgItemMsg()** function may then be used with this handle and the window identifier of the required frame control.

6.6.6 Creating Message Parameters

Before a message can be passed to a target window, its message parameters must be created from the necessary data items. As mentioned in 4.2, "Messages" on page 40, message parameters are 32-bit fields. Presentation Manager provides a number of macros to convert existing data types into the correct representation, and to extract data from message parameters within the target window procedure. These macros are described in Table 3.

Table 3. Presentation Manager Macros. This table shows the "C" language macros provided by Presentation Manager to facilitate the construction and extraction of message parameters.

Macro	Usage
MPFROMP	Produces an MPARAM data type from a pointer
MPFROMHWND	Produces an MPARAM data type from a window handle (HWND)
MPFROMCHAR	Produces an MPARAM data type from an unsigned character (UCHAR)
MPFROMSHORT	Produces an MPARAM data type from a short integer (SHORT or USHORT)
MPFROM2SHORT	Produces an MPARAM data type from two short integers (SHORT or USHORT)
MPFROMSH2CH	Produces an MPARAM data type from a short integer (SHORT or USHORT) and two characters (CHAR or UCHAR)
MPFROMLONG	Produces an MPARAM data type from a long integer (LONG or ULONG)
PVOIDFROMMP	Produces a pointer from an MPARAM data type
HWNDFROMMP	Produces a window handle (HWND) from an MPARAM data type
CHAR1FROMMP	Produces a character (UCHAR) from bits 0-7 of an MPARAM data type
CHAR2FROMMP	Produces a character (UCHAR) from bits 8-15 of an MPARAM data type
CHAR3FROMMP	Produces a character (UCHAR) from bits 16-23 of an MPARAM data type
CHAR4FROMMP	Produces a character (UCHAR) from bits 24-31 of an MPARAM data type
SHORT1FROMMP	Produces an unsigned short integer (USHORT) from bits 0-15 of an MPARAM data type
SHORT2FROMMP	Produces an unsigned short integer (USHORT) from bits 16-31 of an MPARAM data type
LONGFROMMP	Produces an unsigned long integer (ULONG) from an MPARAM data type

For example, to create message parameter *mp1* composed of two unsigned integers *usInt1* and *usInt2*, the following statement is used:

```
mp1 = MPFROM2SHORT(usInt1,usInt2);
```

Similarly, to extract two unsigned integers *usInt3* and *usInt4* from the message parameter *mp2*, the following statements are used:

```
usInt3 = SHORT1FROMMP(mp2);
usInt4 = SHORT2FROMMP(mp2);
```

Characters, pointers, window handles, etc., may all be placed into and retrieved from message parameters using macros supplied by Presentation Manager.

6.6.7 Broadcasting Messages

In certain circumstances, a window procedure may wish to indicate an event to multiple windows, and therefore need to pass the same message to each of these windows. Presentation Manager provides the capability for a message to be broadcast to multiple windows with a single **WinBroadcastMsg()** function call.

The **WinBroadcastMsg()** function passes a message of a specified class to the descendants of a specified parent window, as shown in Figure 35.

```
rc = WinBroadcastMsg(hwnd,                  /* Current is parent       */
                     WMP_MYMESSAGE,         /* Message identifier      */
                     mp1,                   /* 1st message parameter   */
                     mp2,                   /* 2nd message parameter   */
                     BMSG_POST);            /* Post message via queue  */
```

Figure 35. WinBroadcastMsg() Function

The example shown in Figure 35 passes a message of the application-defined class WMP_MYMESSAGE to all children of the current window (that is, the window associated with the window procedure in which the function call is made), with message parameters as shown. The message is posted to the target windows via a message queue, and is thus processed asynchronously; the **WinBroadcastMsg()** function also allows for synchronous processing using the BMSG_SEND flag.

The parent/child hierarchy allows windows to be grouped in particular ways to suit application requirements. For example, all the object windows created by an application may be created as children of a "dummy" master object window. If a particular message must then be sent to all these object windows (for example, to close all the windows), this can be done by broadcasting the message to all children of the master object window.

The BMSG_DESCENDANTS flag may be set in the **WinBroadcastMsg()** call to cause a message to be passed to all descendants of the specified parent window, rather than just the direct children of that parent. This enables a message to be broadcast to a wider target group, should the application so require. Alternatively, the BMSG_FRAMEONLY flag may be set, causing the message to be passed only to frame windows. This is useful in situations where an application wishes to initiate an action by multiple display windows at the same time.

The **WinBroadcastMsg()** function must be used with caution, particularly when it may cause messages to be sent to windows created by other applications. This is possible if the BMSG_DESCENDANTS flag is set and the desktop window is specified as the parent, and may cause complications in other applications. For example, consider the following message definitions:

Application 1

```
#define     WMP_REFRESH     WM_USER+12
```

Application 2

```
#define     WMP_CLOSEALL    WM_USER+12
```

In the example above, each application defines a message class, and each message is to be used for a different purpose. However, both messages have the same message identifier. Now let us assume that Application 1 makes the following function call:

```
rc = WinBroadcastMsg(HWND_DESKTOP,
                     WMP_REFRESH,
                     mp1,
                     (MPARAM)0,
                     BMSG_POST);
```

This function call would cause a WMP_REFRESH message to be passed to all display windows in Application 1 *and* Application 2. However, the windows in Application 2 would interpret the message as a WMP_CLOSEALL message, with possibly undesirable results.

It is therefore strongly recommended that developers exercise extreme care in using the **WinBroadcastMsg()** function, in order to accurately determine the potential results of the messages being broadcast.

6.7 Passing Control

The use of functions and subroutines in an object-oriented application executing in the Presentation Manager raises some issues with regard to object boundaries. In general, the scope of a function or subroutine should be restricted to a single application object, and the processing performed by that subroutine should therefore relate *only* to the data object(s) owned by that application object. If a subroutine invoked from one application object will perform processing on a data object related to a different application object, then the subroutine should be invoked by the second application object, by way of a message passed from the first application object.

Four general types of subroutines may exist within an object-oriented application. These are discussed in the following sections, and are classified according to the nature of their inputs and outputs.

6.7.1 Direct Invocation/Direct Return

This type of subroutine corresponds to the "conventional" subroutine call, in that a parameter list is passed to the subroutine from the calling routine, and a number of parameters and/or a return code is returned at the end of the subroutine's execution. Within an object-oriented application, such subroutines should be used to perform processing that is limited in scope to a single application object (such as an SQL query on a database owned by the application object), or to perform a standard processing function that is common to a number of objects, but where the scope of each execution instance is limited to a single object. For example, a function *DrawCircle* may be called by a number of window procedures to display a circular graphics primitive; however, each invocation of the function is from a single window procedure.

6.7.2 Direct Invocation/Message Return

This type of subroutine should be used where the scope of the processing performed by a subroutine is limited to a single application object, but where the result of that processing must be communicated to an application object other than the one that invoked the subroutine. Since the conventional method of achieving communication between objects is via messages, the subroutine posts a message to the affected application object using the **WinPostMsg()** call, or synchronously passes the message to the application object using the **WinSendMsg()** call (this latter call should be used with caution; see 6.7.3, "Message Invocation/Direct Return"). The message is routed to the destination window by Presentation Manager. The subroutine typically returns to its caller in the normal fashion; this method of passing control is therefore merely a variation of the previously described Direct Invocation/Direct Return method.

Assuming that the calling routine is a window procedure, the return code (if any) from the subroutine is passed to the calling window procedure and that procedure completes its execution before the message resulting from the called subroutine is processed.

Note that this technique can be used where the called subroutine executes in a secondary thread, and the resulting message is passed back to the calling window procedure in the primary thread to indicate the completion of the secondary thread's processing. See Chapter 10, "Multitasking Considerations" for further discussion of multiple threads.

6.7.3 Message Invocation/Direct Return

This type of subroutine occurs with window procedures that are invoked synchronously via a **WinSendMsg()** call. The message is processed, and the return code from the window procedure is routed to the calling routine by Presentation Manager. The calling routine then completes its execution. If any queued messages are generated by the called window procedure or subroutine, these messages are not processed until the calling routine completes its execution and the application issues its next **WinGetMsg()** call. This is so, even if the called window procedure or subroutine executes in a separate thread.

This type of invocation should be used for access to another application object, where the function to be performed must be executed synchronously and the result returned directly to the caller. However, use of the **WinSendMsg()** call in preference to the **WinPostMsg()** call for communication between objects may result in messages being processed out of order due to the application pre-empting the normal order of execution determined by the message queue. The **WinSendMsg()** function should thus be used with care. The use of this call may also extend the time interval between successive **WinGetMsg()** calls to Presentation Manager, thus decreasing the application's responsiveness to user interaction.

6.7.4 Message Invocation/Message Return

This is the case for window procedures invoked in the standard way using a **WinPostMsg()** call from another window or using the **WinDispatchMsg()** function from the application's main routine. In this case the message is processed, and any messages generated during execution are posted to the appropriate queue, but the return code from the window procedure is passed only to Presentation Manager, and does not reach the calling window procedure. For this reason, it is important that any message that requires acknowledgement be handled in

such a way that the window procedure generates a message that is routed to the calling window, and that contains the required acknowledgement.

This type of invocation should be used for access to other application objects, where the function to be performed need not be performed synchronously, and where acknowledgement or completion of the processing may be indicated by a subsequent message posted to the caller.

Note that this should be the default method of invocation for window procedures, since the asynchronous nature of the processing allows the application to maintain the highest level of responsiveness to user interaction.

Note also that a window may receive messages from a number of sources. This allows a window to service requests from a number of other windows, in accordance with a **client-server** architecture. This concept is discussed further in 10.9, "Client-Server Applications" on page 236.

6.7.5 External Macros

An application may also pass control synchronously to an external routine such a macro or subprogram written using the **Restructured Extended Executor** (**REXX**) procedure language. This can be achieved quite easily by calling the REXX command interpreter using the **RexxStart()** function from any point within the application. This function is illustrated in Figure 36.

```
#define INCL_REXXSAA

#include <rexxsaa.h>
   :
PSZ szFileName;                              /* File to be accessed   */
PSZ szOptions;                               /* Command arguments     */

RXSTRING arg;                                /* REXX argument string  */
RXSTRING RexxRetValue;                       /* Result                */

LONG lRexxRC;                                /* REXX return code      */

static RXSYSEXIT ExitList[] = {{"TIXXSIO", RXSIO}, /* Exit handler      */
                               {NULL, RXENDLST}};
   :
arg.strptr = szOptions;                      /* Set argument string   */
arg.strlength = strlen(szOptions);           /* Size of arg string    */

rc = RexxStart(1,                            /* Call REXX           */
               &arg,                         /* Argument string     */
               (PSZ)"RexxProc",              /* REXX proc file      */
               NULL,                         /* Procedure in file   */
               (PSZ)"TIXX",                  /* ADDRESS environment */
               (SHORT)RXCOMMAND,             /* REXX command        */
               (PRXSYSEXIT)ExitList,         /* Exit list routines  */
               &lRexxRC,                     /* Return code address */
               &RexxRetValue);               /* Returned result     */
```

Figure 36. Calling External Macros. This example shows the use of the **RexxStart()** *function to call the REXX command interpreter from within an application.*

Commands are passed to the REXX interpreter using command strings defined using the RXSTRING data type, which is defined in the *rexxsaa.h* header file. This structure contains the string pointer and an unsigned long integer containing the length of the string in bytes. A number of commands may be passed in a single operation, by specifying an array of RXSTRING structures in the second parameter to the **RexxStart()** function. The first parameter specifies the number of commands being passed.

The third parameter to the **RexxStart()** function defines the name of the REXX procedure to be invoked. In Figure 36, the procedure is contained in the file REXXPROC, with an assumed default file extension of .CMD.

If the REXX procedure invoked by the application issues its own commands such as SAY (to output information to the screen), a subcommand handler must be specified in the **RexxStart()** function call, in order to trap such output. A subcommand handler is simply a subroutine which accepts, as parameters, the function and subfunction names issued by the REXX procedure, along with a pointer to an RXSTRING structure which may be used by the subcommand handler to return any information to the REXX procedure. A subcommand handler may reside within the application's main executable module or in a DLL, and must be registered prior to issuing the **RexxStart()** function call, using the **RexxRegisterSubcomExe()** or **RexxRegisterSubcomDll()** functions.

The REXX interpreter's operating environment may be customized through the use of user exits, whereby special routines may be inserted at particular points in the interpreter's execution. Such routines are specified using an array of RXSYSEXIT structures, which identify the exit point and the entry point address of the routine to be invoked at that point. The address of this array is passed in the **RexxStart()** function call.

Use of the REXX interpreter, the **RexxStart()** function and its supporting functions are described in detail in the *IBM OS/2 Version 2.0 Technical Library - Procedures Language/2 REXX Reference.*

6.8 Terminating an Application

A Presentation Manager application is normally terminated by a message of the class WM_QUIT being posted to the application's message queue. The message may be posted by any window procedure or subroutine within the application, or by Presentation Manager as the result of the user selecting the "Shutdown" option from the Presentation Manager desktop. The message may result from the user selecting an "Exit" option from the menu bar, or selecting the "Close" option in the system menu of the application's main window.

The WM_QUIT message causes the next **WinGetMsg()** call to return FALSE. This in turn causes the application's message processing loop to terminate.

By convention, a Presentation Manager application performs standard termination processing such as:

- Destroying the application's main window
- Destroying the application's message queue
- Deregistering the application from Presentation Manager.

The application may additionally perform its own termination functions such as closing or destroying any global data objects. Secondary threads are normally terminated from within the window procedure that created them, as part of that window procedure's WM_DESTROY message processing. See Chapter 10, "Multitasking Considerations" for further information.

6.9 Summary

It can be seen that by making effective use of the facilities provided by Presentation Manager, and by following a number of simple guidelines in the design and implementation of applications, it is relatively simple to develop a Presentation Manager application that conforms to module-based object-oriented programming standards, and achieves benefits through reduced development effort and easier application maintenance, due to code reuse and encapsulation.

It must be accepted however, that some deviation from strict object-oriented practice may be necessary in order to preserve other important goals such as the preservation of responsiveness to the end user. Adherence to academic principles should not take precedence over achievement of the required result.

The mapping of data objects into application objects must be approached with great care in the design stage of a Presentation Manager application. Presentation Manager allows the creation of window procedures (application objects) that operate on more than one data object, and of multiple window procedures that operate on the same data object. This practice should be discouraged however, since it reduces the level of encapsulation in the application object, increases the interdependence between application objects, and consequently reduces the benefits attainable through code reuse and containment of change.

Notwithstanding, the Presentation Manager environment affords great opportunity for the development of applications that implement the general principles of the object-oriented approach. A central precept of object-oriented design is the generic nature and consequent reusability of the objects so created. Adherence to guidelines that promote conformance to object-oriented concepts such as data abstraction, encapsulation and polymorphism, in conjunction with the facilities provided by Presentation Manager for object creation, communication and subclassing, and by the OS/2 operating system in the form of dynamic linking, facilitates the development of highly granular, reusable generic objects in the Presentation Manager environment.

Chapter 7. Workplace Shell and the System Object Model

The Workplace Shell provided under OS/2 Version 2.0 introduces an object-oriented layer into the Presentation Manager environment. It provides a mechanism for the registration of object classes, creation of objects within those classes, and the inheritance of characteristics and behaviors from existing object classes. Using the Workplace Shell, an application may be created as a series of objects that interact on the desktop, and which the user manipulates to perform the required application processing. Each object possesses data, which may be defined for the entire class or for each instance, and a set of methods that operate upon that data.

The Workplace Shell functions that allow the creation and manipulation of objects are based upon the **system object model**, which establishes a basic inheritance hierarchy for objects in the system and defines the underlying protocols which regulate the relationships between objects. The concepts behind the system object model are described in detail in *OS/2 Version 2.0 - Volume 3: Presentation Manager and Workplace Shell*, and *System Object Model Guide and Reference*.

This chapter is an enhanced and expanded version of Chapter 7 in *OS/2 Version 2.0 - Volume 4: Writing Applications*. It adds more detail about areas of WPS/SOM programming, such as Drag/Drop and debugging, and provides a further example Workplace Object to illustrate these techniques. This new version of the chapter is included in both the revised version of *OS/2 Version 2.0 - Volume 4: Writing Applications*, and also in *&volborg.*.

This chapter includes examples of code from two Workplace Objects that have been especially written for this document. They are the pwFolder and pwFinanceFile Workplace Objects. The full source for these and other examples can be found on the diskette included with this document and the program listings are in Appendix E, "Source Code for the PWFolder and PWFinanceFile objects" on page 347.

7.1 Objects in the Workplace Shell

An object in the Workplace Shell conforms closely to the definition of an application object given in Chapter 4, "The Presentation Manager Application Model," in that it consists of a set of data and a number of methods that operate upon that data. Each Workplace Shell object is an instance of a particular object class. In accordance with normal object-oriented theory, the class defines the basic characteristics of the object and the way in which the object responds to events.

7.1.1 Inheritance Hierarchy

Each object class is descended from another class, known as its **parent class**. Since the system object model supports the object-oriented concept of inheritance, a class may inherit data and methods from its parent class, which in turn may inherit data and methods from *its* parent, and so on. A class which inherits properties from other classes is therefore known as a **descendant** of those classes, and the classes from which it inherits are known as **ancestors**. The implementation of inheritance in the Workplace Shell means that when creating a new object class, a programmer simply subclasses the parent class,

and need only define those characteristics that are not defined by, or are different from those of the parent class. This greatly simplifies the process of creating a new object class.

Under the system object model, every object class is a descendant of the base class *SOMObject*. This class defines the basic characteristics and behaviors common to all objects in the system. Other object classes are subclasses of this class. The system object model provides two additional classes, *SOMClass* and *SOMClassManager*, to form the basis of an inheritance hierarchy. The Workplace Shell extends this hierarchy by creating a number of classes of its own, based upon the *SOMObject* class. These Workplace Shell object classes define the characteristics of the object types that are defined and implemented by the Workplace Shell itself.

The inheritance hierarchy implemented by the Workplace Shell is illustrated in Figure 37.

```
SOMObject
 ├─SOMClass
 ├─SOMClassManager
 └─WPObject
    │   ├──WPAbstract
    │   │    ├─WPClock
    │   │    ├─WPCountry
    │   │    ├─WPKeyboard
    │   │    ├─WPMouse
    │   │    ├─WPPalette
    │   │    │    ├──WPColorPalette
    │   │    │    ├──WPFontPalette
    │   │    │    └──WPSchemes
    │   │    ├─WPPrinter
    │   │    ├─WPPower
    │   │    ├─WPProgram
    │   │    ├─WPShadow
    │   │    ├─WPShredder
    │   │    ├─WPSound
    │   │    ├─WPSpecialNeeds
    │   │    ├─WPSpooler
    │   │    └─WPSystem
    │   ├──WPFileSystem
    │   │    ├─WPDataFile
    │   │    ├─WPFolder
    │   │    │    ├──WPDesktop
    │   │    │    ├──WPDisk
    │   │    │    ├──WPDrives
    │   │    │    ├──WPStartup
    │   │    │    └──WPTemplates
    │   │    └─WPProgramFile
    │   └──WPTransient
    │        ├─WPJob
    │        ├─WPPort
    │        ├─WPPrinterDriver
    │        └─WPQueueDriver
```

Figure 37. Workplace Shell Inheritance Hierarchy

As well as being descended from the system object model base inheritance hierarchy, all Workplace Shell object classes are descended from one of three **base storage classes** defined by the Workplace Shell. These classes are so named because they directly influence the storage of control information and instance data for the class. The three predefined base storage classes are:

- **WPAbstract**, which is the base class for abstract objects such as programs, devices, etc., and for which control information is stored in the system initialization file OS2.INI.
- **WPFileSystem**, which is the base class for objects that are stored as files in the file system, and for which control information is stored in the file system as extended attributes.
- **WPTransient**, which is the base class for objects that only exist during execution of a particular program; that is, the object is created and used for a particular purpose during processing, and then immediately deleted from the system.

An application developer may extend the Workplace Shell inheritance hierarchy by introducing new object classes based upon those already implemented by the Workplace Shell itself. Indeed, the developer may even introduce new base classes, although this is definitely a non-trivial exercise and should be approached with caution.

7.1.2 Metaclasses

Just as each Workplace Shell object is an instance of a class, the class itself is an instance of another class known as its **metaclass**. Just as an object has instance data and methods that pertain only to a specific instance of the class, so the metaclass has class data and methods that pertain to the *entire* class. Such methods are known as **class methods**, whereas methods that operate only for a particular instance of the class are known as **instance methods**.

Class methods and data are available to the programmer when creating new object classes. A programmer may introduce new class data and methods for an object class, as well as instance data and methods. Similarly, a new object class may override existing class methods to modify the processing performed by those methods.

7.1.3 Class Implementation

Each object class in the Workplace Shell resides in a dynamic link library (DLL). A programmer creates an object class by defining its characteristics in a **class definition file**. This file is then used as input to the **SOM Precompiler**, in order to produce "C" source code and header files for the object class. This source code includes basic definitions for the object class's data and methods; the code is then edited by the programmer to include the logic for each of the required methods. Once the code is complete, it is compiled and link edited in the normal way to produce a dynamic link library; see OS/2 2.1 Volume 4: Writing Applications, Chapter 14 "Compiling and Link Editing an Application" for further information on compiling and link editing.

When an object class has been created, it must be registered with the Workplace Shell, which includes the DLL in a list of libraries loaded at initialization time. The entry points for the DLL are known to the Workplace Shell, and may be called in order to invoke the object's methods.

The process of creating an object class from a class definition file is described in 7.3, "Defining an Object" on page 114.

7.2 Object Structure

In the simplest case, an object in the Workplace Shell consists of methods and instance data. The PM Window Manager communicates events to the Workplace Shell using messages, which in turn invokes the object's methods to perform the processing indicated by the event. This is in accordance with the definition of an application object given in *OS/2 2.1 Volume 4: Writing Applications, Chapter 4 "The Presentation Manager Application Model"*. Note that since the Workplace Shell provides a more extensive inheritance hierarchy than the base Presentation Manager application model, the method invoked by a particular message may belong explicitly to the object in question, or may belong to its parent (and be inherited from that parent).

The structure of an object in the Workplace Shell is therefore very similar to that of a window in the conventional Presentation Manager application model; the Workplace Shell object simply takes the object-oriented concepts to a higher degree of implementation. Therefore the constructs implemented by Presentation Manager under previous versions of OS/2 can often be implemented more elegantly with the Workplace Shell.

For the remainder of this chapter, two examples are used to explain the structure and behavior of an object class. These are the pwFolder and pwFinanceFile Workplace Objects.

- The pwFolder Workplace Object is a specific type of Workplace Shell folder which has a password defined so that it can be locked to prevent access by an unauthorized user. This object class is implemented by subclassing the *WPFolder* class to create a new object class named *PWFolder*, adding new methods and overriding existing methods where appropriate.
- The pwFinanceFile Workplace Object is a specific type of Workplace Shell data file which has a password defined so that it can be locked to prevent access by an unauthorized user. Additional methods have been added to provide specific behavior and this is covered later in this chapter. This object class is implemented by subclassing the *WPDataFile* class to create a new object class named *PWFinanceFile*, adding new methods and overriding existing methods where appropriate.

Sample code is provided in the text, and on an included diskette, for the various methods used to add the password protection to the folder.

7.2.1 Methods

In a Presentation Manager application, a window procedure receives messages from Presentation Manager, determines the type of message and invokes a series of program statements (which effectively constitute a method) as a result of that message. A Workplace Shell object operates in a similar fashion, except that the Workplace Shell itself determines the type of message and invokes the corresponding method, without any explicit action on the part of the object.

Therefore, whereas the Presentation Manager window procedure comprises a *case* statement with each case being a method, the Workplace Shell object eliminates the need for the *case* statement and allows the Workplace Shell to invoke the methods directly. The syntax for invoking a method from within an

object or application is hence very similar to that for invoking a subroutine; the only real difference is that a method may be accessed from outside the object itself (that is, from another object or from an application), while a subroutine is normally private to the object.

Many methods are defined by the *WPObject* class, from which application-defined classes are typically descended. When creating a new object class, a programmer may override the methods already defined by the class's ancestors, and/or include new methods specific to the class being created. The methods defined by the *WPObject* class are described in the *IBM OS/2 Version 2.0 Presentation Manager Reference*. Programmers who wish to create new object classes descended from this case should read the descriptions of these methods to determine the extent of the modifications necessary.

7.2.1.1 Invoking a Method

As mentioned in *OS/2 2.1 Volume 4: Writing Applications, Chapter 4 "The Presentation Manager Application Model"* , methods within an object are invoked as a result of messages that communicate events to the object. These events may be initiated by the user (for example, as a result of clicking the mouse on an object's context menu), by the object itself or another object, or by the system to indicate a system event such as opening or closing a view of the object.

The syntax for invoking a method is similar to that for invoking a subroutine, with one exception. The first parameter passed in the call is a pointer to an object that is capable of invoking the method called the "receiver", and this is typically a pointer to the object itself. This is illustrated in Figure 38, where a sample invocation of a method named _wpSetTitle is shown.

```
PWFolder *somSelf;                          /* Pointer to self      */
PSZ      szTitle;                           /* Title string         */

_wpSetTitle(somSelf,szTitle);               /* Set title string     */
```

Figure 38. Invoking a Method

The _wpSetTitle method is defined by the *WPObject* class, and is inherited by all classes descended from the class. The method accepts a title string and sets the title of the object; that is, the text that appears below the object's icon on the Workplace Shell desktop.

The pointer *somSelf* is defined by the SOM Precompiler when it creates the "C" source code from the class definition file. In the example above, *somSelf* is defined as a pointer to an object of class *PWFolder* and within a method, allows the method to access the instance data of the object to which it belongs. The need to pass this pointer arises from the limitations of the "C" language syntax under which the current implementation of the Workplace Shell operates; other languages such as C++ may be able to invoke methods in a more elegant manner.

7.2.1.2 Method Processing and Instance Data

Within a method, the *somSelf* pointer, passed as the first parameter in the call to the method, acts as a pointer to the method's own object, and allows the method to access its instance data. The SOM Precompiler automatically provides a base pointer named *somThis* that references the instance data, and includes a call to a method that initializes this pointer from the object pointer:

```
PWFolderData *somThis = PWFolderGetData(somSelf);
```

When this statement has successfully executed upon entry to the method, the method has access to the object's instance data. For example, the password-protected folder has a password string, which may be accessed by a method using the following name:

```
somThis->szPassword
```

To make things simpler, the SOM Precompiler generates a macro for each instance variable, in a manner similar to that used for function names:

```
#define _szPassword (somThis->szPassword)
#define _szCurrentPassword (somThis->szCurrentPassword)
#define _szUserid (somThis->szUserid)
```

This macro is included in a header file for the object class, and avoids the need for the programmer to type the complete name throughout the source code.

Once the instance data is available to the method, any application logic may be performed, including the use of OS/2 and Presentation Manager resources. See 7.4.5, "Accessing Presentation Manager Resources From a Workplace Shell Object" on page 148 for additional considerations on the use of Presentation Manager resources from within a Workplace Shell object.

7.2.1.3 Returning from a Method

In order to return control to its calling routine, a method simply uses the *return* statement. Any valid form of return code may be passed to the calling routine as a parameter to this statement, provided that the data type of the return code is consistent with the declaration of the method. The data type of the return code is typically set by the SOM Precompiler, and a default *return* statement provided, based on information supplied by the programmer when the method is defined in the *Methods* section of the class definition file (see 7.3.2, "Class Definition File" on page 115).

7.2.1.4 Overriding Existing Methods

A new object class may override one or more of the existing methods defined by its parent class, either to completely replace the processing performed by these methods, or to add its own processing to that already performed by the parent. An example of an object class overriding the _wpSetTitle method is shown in Figure 39 on page 107.

```
SOM_Scope BOOL SOMLINK pwfolder_wpSetTitle(PWFolder *somSelf,
                                           PSZ pszNewTitle)
{
    CHAR szBuf[100];                          /* Character buffer      */

    PWFolderData *somThis =                   /* Get instance data     */
                PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",           /* Set debug info        */
                        "pwfolder_wpSetTitle");

    strcpy(szBuf,pszNewTitle);                /* Get current title     */

    if ((strcmp(_szCurrentPassword,           /* If folder is locked   */
                _szPassword)) != 0)
        if((strstr(szBuf,"LOCKED")) == NULL )  /* and <LOCKED> not in   */
                                              /* current title         */
          strcat(szBuf," <LOCKED>");          /* Add <LOCKED> to title */

    return(parent_wpSetTitle(somSelf,         /* Allow default proc to */
                             szBuf));         /* occur                 */
}
```

Figure 39. Overriding an Existing Method. This example shows the _wpSetTitle method being overridden to add the word "LOCKED" to the end of the title of a locked password-protected folder.

The example given in Figure 39 shows the use of class-specific processing to modify the title of a password-protected folder. The inclusion of the string "<LOCKED>" at the end of the user-specified title provides a visual indication to the user that the folder is locked. Additional visual indication is provided by modifying the icon when the folder is in the locked state; the code that carries out this operation is included in the *_LockFolder* method shown in Figure 40 on page 108.

The strings *_szCurrentPassword* and *_szPassword* are instance data items defined by the new object class. These data items are actually accessed using the *somThis* pointer; however, the SOM Precompiler defines a macro for each instance data item, as described in 7.2.1.2, "Method Processing and Instance Data" on page 106.

Note that most workplace methods require that parent processing be performed during the override function. Normally this would be part of the return statement, but some methods require parent processing to be done first. You should check the method description to determine where the parent processing needs to be done.

7.2.1.5 Adding New Methods

In addition to overriding existing methods defined by the parent class, an object class may also add new methods to carry out processing for events not handled by the parent class. For example, the password-protected folder example must have a mechanism to lock the folder. This is implemented as a new method named *_LockFolder*, as shown in Figure 40 on page 108.

```
SOM_Scope BOOL SOMLINK pwfolder_LockFolder(PWFolder *somSelf)
{
    HPTR  hLockedIcon;

    PWFolderData *somThis =                     /* Get instance data     */
                PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",             /* Set debug info        */
                        "pwfolder_LockFolder");

    strcpy(_szCurrentPassword,                  /* Invalidate current    */
           "NOPASSWD");                         /* password              */
    _wpSetTitle(somSelf,                        /* Set title             */
                _wpQueryTitle(somSelf) );

    hLockedIcon = WinLoadPointer(HWND_DESKTOP,  /* Load "lock" icon      */
                                 (HMODULE)0,
                                 LOCK);

    _wpSetIcon(somSelf,                         /* Set icon to locked    */
               hLockedIcon);                    /* appearance            */

    return((BOOL)0);                            /* Return                */
}
```

Figure 40. Adding a New Method

This method simply copies a default string to the variable *_szCurrentPassword* that contains the last supplied password entry from the user, so that when a comparison is made between this variable and the folder's password, the two do not match. This effectively locks the folder and prevents any view of it being opened. To provide a visual indication to the end user that the folder is locked, a "locked" icon is loaded using the Presentation Manager **WinLoadPointer()** function, and the _wpSetIcon method is invoked to set this as the folder's new icon on the desktop.

Note that the definition for adding a new method is very similar to that for overriding an existing method. The primary difference is that, since the new method is specific to the object class and is not defined by the parent class, there is no need to invoke the parent class's method to perform default processing for the method.

7.2.1.6 Attaching a Method to the Context Menu

A method may be invoked as a result of the user selecting an item from the object's context menu. In order to allow this, an item must be added to the context menu, and an appropriate action must be taken by the object when that item is selected by the user.

An item can be added to the context menu for an object class by overriding the _wpModifyPopupMenu method defined by the *WPObject* class, and including a call to the _wpInsertPopupMenuItem method to insert the item. This technique is shown in Figure 41 on page 109.

```
#define MI_LOCK     WPMENUID_USER+1
   :
   :
SOM_Scope BOOL SOMLINK pwfolder_wpModifyPopupMenu(PWFolder *somSelf,
                                                  HWND hwndMenu,
                                                  HWND hwndCnr,
                                                  ULONG iPosition)
{
    PWFolderData *somThis =                    /* Get instance data     */
                 PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",            /* Set debug info        */
                        "pwfolder_wpModifyPopupMenu");

    _wpInsertPopupMenuItems(somSelf,           /* Insert menu item      */
                            hwndMenu,          /* Menu handle           */
                            iPosition,         /* Default position      */
                            hModule,           /* Module handle         */
                            MI_LOCK,           /* Menu item id          */
                            0);                /* No submenu id         */

    return(parent_wpModifyPopupMenu(somSelf,   /* Allow default proc to */
                                    hwndMenu,  /* occur                 */
                                    hwndCnr,
                                    iPosition));
}
```

Figure 41. Adding an Item to a Context Menu

The example shown in Figure 41 adds a *Lock* item to the context menu for the password-protected folder object. This allows the folder to be locked by the user at any time, irrespective of whether a view of the folder is currently open.

The _wpInsertPopupMenuItem method adds a menu item or a submenu to the existing context menu for the object. The item identifier for the menu item or submenu (MI_LOCK in the above example) is an integer constant that is typically defined in the header file. Note that the value of this constant should be specified as an offset from the system-defined constant WPMENUID_USER, rather than an absolute integer value. Following this convention will avoid any clashes with item identifiers defined by the Workplace Shell for default context menu items.

Since the password-protected folder is a descendant of the *WPFolder* class defined by the Workplace Shell, the default context menu items for the *WPFolder* class should also appear. The default processing for the parent class is therefore invoked as part of the _wpModifyPopupMenu processing for the new object class.

Once the required item is added to the context menu, the object must be able to detect when the item is selected in order to invoke the appropriate method. By default, the _wpMenuItemSelected method is invoked by the system whenever the user selects an item from the context menu. This method, which is defined by the *WPObject* class, may be overridden by a new object class in order to check for the presence of a new item and invoke the appropriate method. The item identifier of the selected item is passed as a parameter to the _wpMenuItemSelected method, and is normally interrogated using a *case* statement, as shown in Figure 42 on page 110.

```
SOM_Scope void SOMLINK pwfolder_wpMenuItemSelected(PWFolder *somSelf,
                                                   HWND hwndFrame,
                                                   ULONG MenuId)
{
    PWFolderData *somThis =                    /* Get instance data     */
                 PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",            /* Set debug info        */
                        "pwfolder_wpMenuItemSelected");

    switch (MenuId)                            /* Switch on item id     */
           {
           case MI_LOCK:                       /* If "Lock" item        */
                _LockFolder(somSelf);          /* Lock folder           */
                break;
           default:                                 /* else             */
                parent_wpMenuItemSelected(somSelf,  /* Allow default    */
                                          hwndFrame, /* processing to   */
                                          MenuId);   /* occur           */
                break;
           }
    return;
}
```

Figure 42. Invoking a Method via a Context Menu Item

The _wpMenuItemSelected method consists of a *case* statement that determines the item selected from the context menu. In the above example, an explicit case is included only for the MI_LOCK item defined by this class. All other menu items are defined by the parent class, and their selection is therefore handled by allowing the parent class's default processing to occur.

7.2.1.7 Modifying the Standard Context Menu Items

The _wpFilterPopupMenu method can be used to filter out (remove) standard menu items that are inherited from the ancestor classes, or to reinstate any of the standard pop-up menu items. This method can also be used to determine if the ancestor classes have filtered out any of the Workplace Shell-provided standard menu items, by checking to see if any of the flags associated with the menu items are not set.

The *ulFlags* parameter of "C" type *ULONG* is really a bit array which is binary *OR*ed together with the ancestor classes *ulFlags* when the parent method is called, effectively adding these menu items together. The resultant *ulFlags* is then returned from the _wpFilterPopupMenu method.

But if the parent method is called first, then the resultant flags are binary *AND*ed with the complement of the menu item (flag) to be removed. Upon returning this from the object's _wpFilterPopupMenu, the item will now be removed from the pop-up menu.

To determine if a menu item is present or not, first call the parent method and then simply binary *AND* the menu item flag with the parent method result. If the result of this operation is the menu item flag that was *AND*ed, then the flag has been set by the ancestor classes; otherwise it has been removed.

Figure 43 on page 111 shows how to test for a menu item, removing the menu item if it is present, or adding it if the ancestor classes removed it. In this case the *Create another* menu item is the menu item of interest.

```
SOM_Scope ULONG   SOMLINK pwFinanceFile_wpFilterPopupMenu(PWFinanceFile *somSelf,
          ULONG ulFlags,
          HWND hwndCnr,
          BOOL fMultiSelect)
{ ULONG ulPopupFlags;

    PWFinanceFileData *somThis = PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile","pwFinanceFile_wpFilterPopupMenu");

    /* first find out what our ancestors have done! */
    ulPopupFlags = parent_wpFilterPopupMenu(somSelf,ulFlags,hwndCnr,fMultiSelect);

    /* now what has been done to the "Create another" menu item */
    if ((ulPopupFlags & CTXT_NEW) == CTXT_NEW) {

       /* the "Create another" menu item is on our Popup, so remove it */
       ulPopupFlags = ulPopupFlags & ~CTXT_NEW;
    } else {

       /* the "Create another" menu item is NOT on our Popup, so add it */
       ulPopupFlags = ulPopupFlags | CTXT_NEW;
    } /* endif */

    return(ulPopupFlags);
}
```

Figure 43. Filtering the Pop-up Menu Items

7.2.1.8 Class Methods

Most object methods are instance methods; that is, they act upon one particular instance of an object class, rather than upon all instances of the class.

However, there are times when it is useful to have methods that operate on the object class itself. These methods may operate on class data rather than instance data, thereby affecting the entire class rather than a single instance of the class. Such methods are known as class methods. The class method _wpclsQueryTitle is defined by the *WPObject* class, and is overridden in the password-protected folder example. An example of the overridden _wpclsQueryTitle method is given in Figure 44 on page 112.

```
PSZ szDefaultClassTitle = "Password Folder";

/*
 *
 *  METHOD: wpclsQueryTitle                                PUBLIC
 *
 *  PURPOSE:
 *    Return the string "Password Folder"
 *
 */
#undef SOM_CurrentClass
#define SOM_CurrentClass M_PWFolderCClassData.parentMtab
SOM_Scope PSZ SOMLINK pwfoldercls_wpclsQueryTitle(M_PWFolder *somSelf)
{
    /* M_PWFolderData *somThis = M_PWFolderGetData(somSelf); */
    M_PWFolderMethodDebug("M_PWFolder","pwfoldercls_wpclsQueryTitle");

    return(szDefaultClassTitle);
}
```

Figure 44. Class Method Example. This example shows an overridden class method _wpclsQueryTitle, which is modified to supply a default title for an object within the class.

The purpose of this class method is to provide the password-protected folder with a default title. This is the title that will appear with the folder's template icon in the *Templates* folder, and which is given to any instances of the class that are instantiated without a title. Since the default title applies to all instances of the class, it is implemented in a class method rather than an instance method.

The prefix "M_" denotes the metaclass in the SOM-generated "C" source. As already mentioned, the first parameter passed to a method is a pointer to a type of object that can invoke that method; this is true for both instance methods and class methods; for a class method the first parameter contains a pointer to an instance of the metaclass.

```
            Pointer to instance of metaclass
            which is a class object ──────┐
                                          ▼
   pwfoldercls_wpclsQueryTitle(M_PWFolder *somSelf)
                               ▲
        Type is Metaclass ─────┘
```

Since a class is also an object, it follows that the class itself has its own "instance data"; hence the next line of code appears as follows:

```
/* M_PWFolderData *somThis = M_PWFolderGetData(somSelf); */
```

This statement would access the SOM object's class data. However, since no class data is specified in the .CSC file, there is nothing to access and so the SOM Precompiler has commented the line out to reflect this.

For simple examples, it is easier to use global variables in the DLL for class data. This technique has been used in Figure 44; the default title string is stored at the beginning of the program into the global variable *szDefaultTitle*. However,

using this technique means that class data can be accessed by instance methods, which is never desirable, and may have adverse consequences, although these may generally be avoided by sound programming techniques.

7.2.1.9 Invoking Another Object's Methods

An object may invoke a method in another object class. This technique is useful in a client-server situation, where one object creates another object of a different class and then wishes to have that object perform certain actions. The system object model provides programming functions that can be used to determine the necessary information and invoke the method. An example is given in Figure 45.

```
SOMAny *RecordClass;                          /* Class object pointer  */
somID  idQueryMethod;                         /* Method id             */

CHAR   szQueryBuffer[100];                    /* Query data buffer     */
PVOID  pFindData;                             /* Returned data buffer  */
   :
   :
rc = DosAllocSharedMem(&pFindData,            /* Alloc shared memory   */
                       NULL,                  /* No name               */
                       sizeof(szQueryBuffer)+1, /* Size of memory object */
                       OBJ_GIVEABLE  |        /* Make object giveable  */
                       PAG_WRITE     |        /* Allow write access    */
                       PAG_READ      |        /* Allow read access     */
                       PAG_COMMIT);           /* Commit storage now    */

strcpy(pFindData,szQueryBuffer);              /* Copy data to buffer   */

RecordClass = _somFindClass(SOMClassMgrObject, /* Get class obj pointer */
                            SOM_IdFromString("Record"),
                                          1,1));
idQueryMethod = SOM_IdFromString("clsQuery");  /* Get method id        */

_somDispatchL(RecordClass,                    /* Invoke method         */
              idQueryMethod,                  /* Method id             */
              (void *)0,                      /* No descriptor string  */
              pFindData,                      /* Method parameters     */
              somSelf);
```

Figure 45. Invoking a Method in Another Object Class

The example given in Figure 45 shows part of a "database client" object that sends a database query to a "database server" object. The client first allocates a shared memory object into which it loads the query. The client then uses the *_somFindClass* method and the *SOM_IdFromString* macro to determine the object pointer for the object, and the method identifier for the required method. The *_somDispatchL* method is then used to invoke the method.

It is also possible to invoke a class method using the object pointer to that class, obtained using the *_somFindClass* method shown in Figure 45. This requires the header file for the class to be included in the source code for the class that will invoke the method, using a *#include* statement. In the module definition file for the invoking class, the following IMPORT statements must be provided:

```
IMPORTS
   record.RecordCClassData
   record.RecordClassData
   record.RecordNewClass
   record.M_RecordCClassData
   record.M_RecordClassData
   record.M_RecordNewClass
```

When these steps have been carried out, a method in the other class may be invoked directly, as follows:

```
_clsQueryDatabase(RecordClass,          /* Invoke class method   */
                  pQuery,               /* Method specific       */
                  Folder);              /* parameters            */
```

While this technique is less clean than the previous approach since it requires the inclusion of the header file and import statements, it provides better performance.

7.2.2 Subroutines

Subroutines may be accessed from within a Workplace Shell object, in much the same manner as from any other program. Normal programming language calling conventions are used. Subroutines used by the object may reside within the same DLL as the object itself, or may be in a different DLL.

A number of guidelines for the use of subroutines within Presentation Manager applications are given in *OS/2 2.1 Volume 4: Writing Applications, Chapter 4 "The Presentation Manager Application Model"*. Note that similar guidelines apply to the use of subroutines within Workplace Shell objects, since these objects should also adhere to object-oriented programming principles.

7.3 Defining an Object

The definition of an object is achieved using a language known as the **Object Interface Definition Language**. The statements that define an object class are entered into the class definition file for the class, which is an ASCII file and may thus be created using any normal text editor. The class definition file is used as input to the SOM Precompiler, which will generate a number of files from the class definition file.

7.3.1 Files

The SOM Precompiler generates a number of files that are used to define an object class to the Workplace Shell and to other classes that may wish to inherit the characteristics and behaviors of the class. These files are:

.H A public header file for programs that use the class.

.PH A private header file, which provides usage bindings to any private methods implemented by the class.

.IH An implementation header file, which provides macros, etc., to support the implementation of the class.

.C A template C file, to which code may be added to implement the class.

.SC A language-neutral class definition.

.PSC A private language-neutral core file, which contains private parts of the interface for the class.

.DEF An OS/2 DLL module definition file containing the relevant exports need to implement the class.

These files may then be used as input to a *C* compiler, generating object code that is in turn linked to create a dynamic link library, which implements the object class.

7.3.2 Class Definition File

The class definition file contains all the information necessary to implement a new class. The file is divided into the following sections:

1. Include section
2. Class section
3. Parent Class section
4. Release Order section
5. Metaclass section
6. Passthru section
7. Data section
8. Methods section

Each of these sections is described in more detail below, using examples from the password-protected folder class described earlier in this chapter.

7.3.2.1 Include Section

Since all system object model classes have a parent, it is necessary to know the name of the parent class and the location of its interface definition. The include section specifies the location of the interface definition file for the parent. In the folder example, only a single line is included:

```
#
# Include the class definition file for the parent class
#
include <wpfolder.sc>
```

Since the folder example is simply a specialized form of the *WPFolder* class, it uses this class as its parent and inherits much of its behavior from the *WPFolder* class. The include section therefore specifies the interface definition for the *WPFolder* class. A full list of Workplace Shell classes and their definition files can be found in the *IBM OS/2 Version 2.0 Presentation Manager Reference*.

Note that the comments that start with a "#" are discarded by the SOM Precompiler; hence the comment in the example above will not be seen in the SOM Precompiler-generated files.

7.3.2.2 Class Section

This section provides basic information about the new class, specifying its name and various attributes. The password folder example has the following class section entry:

```
#
#   Define the new class
#
class: PWFolder,
       file stem = pwfolder,
       external prefix = pwFolder_,
       class prefix = pwFoldercls_,
       major version = 1,
       minor version = 1,
       local;
-- PWFolder is a Password-protected folder.
-- Its derived as follows:
--        SOMObject
--          - WPObject
--            - WPFileSystem
--              - WPFolder
--                - PWFolder
```

All class definition files must contain a class section. Certain statements within the class section are mandatory, while others are optional.

The first item in the class section is a *name*:

```
class: PWFolder,
```

All classes must have a name.

The *file stem* specifies the file name to be used by the SOM Precompiler for the generated files. For example, if the file stem statement reads:

```
file stem = myfile
```

then the .DEF file generated by the SOM Precompiler would be called *myfile.def*.

The *external prefix* specifies a prefix to be used by the SOM Precompiler on all function names. Hence if an external prefix of "pwFolder_" is specified and a method is named "SetInfo," the function name generated by the SOM Precompiler would be "pwFolder_SetInfo."

The SOM Precompiler normally generates a macro for all methods defined by the class, such that the method is referenced in the source code by its defined name, preceded by an underscore character. For example, the method *pwFolder_SetInfo* described above would be referenced simply as *_SetInfo*. This helps make the source code more readable and avoids the need for the programmer to type the full name when editing the code.

The *class prefix* is similar to the external prefix, except that it is used specifically for functions that are class methods. The differences between class methods and instance methods are discussed in 7.2.1.8, "Class Methods" on page 111.

The *major version* and *minor version* are used to ensure that the bindings are at the right level for the class implementation code.

The *local* option is used to specify that binding files should be linked locally. In "C" programming terms, this means that the following source code is generated:

```
#include "wpfolder.h"
```

If the *global* option is used, the resulting source code would be as follows:

```
#include <wpfolder.h>
```

The last part of the class section is for comments. Using "--" as the comment style causes a comment block to be passed through to the interface definition (.SC) file.

7.3.2.3 Parent Class Section

The parent class section specifies the parent of the new class. All classes must have this section. The parent class section for the password-protected folder example appears as follows:

```
#
# Parent class
#
parent: WPFolder;
```

7.3.2.4 Release Order Section

This section allows the programmer to specify the sequence in which the methods and public data will be released. Since this sequence is maintained by the SOM Precompiler, other programs using this class will not need to be recompiled every time something new is added to the class.

Note that for future compatibility it is essential that all public and private methods are listed in the release order section, and their order does not change. It is strongly suggested that the "-r" option be used with the SOM compiler to produce any release order warnings.

The password-protected folder example has only one public method in addition to those already defined by its ancestor classes. This method is seen in the release section as follows:

```
#
# Specify the release order of new methods
#
release order: LockFolder;
```

Since other public methods are defined by the parent class or by its ancestors, the programmer creating an object class need not define these methods in the class definition file. Hence the programmer need not be aware of the existing methods in the parent class, unless they require modification for use by the new class. This is in accordance with the object-oriented concept of encapsulation.

7.3.2.5 Metaclass Section

For the password-protected folder example (and in most other cases) an explicit metaclass is not required. The concept of metaclasses is discussed in 7.1.2, "Metaclasses" on page 103. Readers desiring more knowledge of programming using metaclasses should refer to the *IBM SOM Programming Reference*.

7.3.2.6 Passthru Section

This section allows the programmer to define blocks of C source code that are passed through to any of the files generated by the SOM Precompiler. Each passthru block is distinguished by an identifier, the syntax of which is as follows:

```
passthru: <language>.<suffix>
```

The password-protected folder example has two passthru sections. The first passthru is "C.h," which passes the code block to the C binding file *pwfolder.h*. This block of code defines a *DebugBox* macro, which can be used anywhere in the code for the new class.

```
#
# Passthru a debug message box to the .ih file
# (for inclusion in the .c file)
#
passthru: C.h, after;

#define DebugBox(Title, Text) WinMessageBox(HWND_DESKTOP,
                                            HWND_DESKTOP,
                                            (PSZ)Text,
                                            (PSZ)Title,
                                            0,
                                            MB_OK |
                                            MB_INFORMATION)

endpassthru;
```

The second passthru block is "C.ph"; this passes the code block to the C binding file *pwfolder.ph*. This block is used to define a data structure that is accessed by the private methods *_GetInfo* and *_SetInfo*, and is used to pass information to and from the dialog procedure that prompts the user for the folder password.

```
#
# Passthru private definitions to the .ph file
# (for inclusion in the .c file)
#
passthru: C.ph;

typedef struct _PWF_INFO {
                        CHAR   szPassword[20];
                        CHAR   szCurrentPassword[20];
                        CHAR   szUserid[20];
                        } PWF_INFO;
typedef PWF_INFO *PPWF_INFO;

endpassthru;
```

7.3.2.7 Data Section

This section lists the instance variables used by the class. In the password-protected folder example, three variables are defined as follows:

```
#
# Define instance data for the class
#
data:
CHAR szPassword[20];
--  This is the password that locks the folder
CHAR szCurrentPassword[20];
--  This is the password the user has entered to be
--  checked against the lock password
CHAR szUserid[20];
--  The userid data is here for future expansion
```

Note that the *szUserid* instance variable is not used in the version discussed in this document, since the current example assumes only a single workstation user. However, it is feasible for user identification to be obtained at startup, and held by the system for authentication against a password to determine whether access is permitted.

7.3.2.8 Methods Section

The last section in the class definition file contains a list of all the methods to be defined by the object class. ANSI C function-prototype syntax is used to define each method. When coding these definitions, it is recommended that the methods be divided into the following parts:

1. Methods that are new for this class
2. Methods that are overridden from ancestor classes

The following section shows two methods taken from the folder example's class definition file.

This first method will be used in the password dialog to take a copy of the object's instance data and place it in a structure that the dialog code may access.

```
#
# Define new methods
#
methods:

BOOL QueryInfo(PPWF_INFO pPWFolderInfo), private;
--
-- METHOD:   QueryInfo                                    PRIVATE
--
-- PURPOSE:  Copy the PWFolder instance data into
--           the PWF_INFO structure that pPWFolderInfo
--           points to.
--
```

The second example shows an overridden method. This method originates in the *WPObject* class, which is a base class. It is used to set up the password string when the folder object is created.

```
#
# Specify methods being overridden
#

override wpSetup;
--
-- OVERRIDE: wpSetup                                     PUBLIC
--
-- PURPOSE:  Here we can set the folder password
--           to that passed in from the object
--           create command.
--
```

More detailed information on class definition files and the OIDL is given in the *IBM SOM Programming Reference*.

7.3.3 C Implementation of an Object Class

When the SOM Precompiler has been run successfully against a class definition file, it will produce all the source files necessary to build a Workplace Shell DLL. The most important of these files for the C programmer is the C source code file, which has an extension of .C. This file contains definitions and "function stubs" for all the methods defined by the class. This file must be edited by the programmer to add the actual application logic to each method. Figure 46 on

page 120 shows the SOM Precompiler-generated function stub for the *QueryInfo* method from the folder example.

```
/*
 *
 *  METHOD:   QueryInfo                                    PRIVATE
 *
 *  PURPOSE:  Copy the PWFolder instance data into
 *            the PWF_INFO structure that pPWFolderInfo
 *            points to.
 *                                  2
 */                                                        3

SOM_Scope BOOL     SOMLINK pwFolder_QueryInfo(PWFolder *somSelf,
                                  PPWF_INFO pPWFolderInfo)
{        1
    PWFolderData *somThis = PWFolderGetData(somSelf);   4
    PWFolderMethodDebug("PWFolder","pwfolder_QueryInfo");   5

    <application logic>   6

    return((BOOL)0);
}            7
```

Figure 46. A SOM Precompiler-generated Function Stub

Notes:

1 SOM_Scope declares the function scope according to the language being used. For example, in C++, SOM_Scope would be defined as *extern C* but in C it is simply defined as *extern*.

2 It can be seen that the external prefix "pwfolder_," which was specified in the class definition file, has been placed in front of the function as expected. Note that the SOM Precompiler generates a macro for this function in the private header file:

```
#define _QueryInfo PWFolder_QueryInfo
```

This avoids the necessity for the programmer to type the full function name, and helps make the code more readable.

3 Since SOM uses the C language, methods from SOM objects cannot be referenced in a very elegant manner. The first parameter to a SOM method must be a pointer to an object that can invoke that method. In the actual method function, this pointer is given the name *somSelf*. For example, the difference between C and C++ is as follows:

```
/* Let us say */

     pMyObject =  (pointer to an object);

// in C++ the following syntax may be used

     pMyObject->Method(param1, param2....);

/* but in C the following is required */

     Method(pMyObject, param1, param2....);
```

4 This statement uses the pointer to the object to initialize a pointer to access the object's instance data. See 7.2.1.2, "Method Processing and Instance Data" on page 106 for further information on instance data.

5 This line will perform tracing. Tracing is switched on whenever the SOM global variable *SOM_TraceLevel* is set to a non-zero value.

6 This section is left blank by the SOM Precompiler for the developer to fill with the application logic. This logic may include access to system and/or Presentation Manager resources. For the password-protected folder example, the *_QueryInfo* method must copy the instance variables to the PWF_INFO data structure defined in the passthru section of the class definition file. The code required to do this is as follows:

```
strcpy(pPWFolderInfo->szPassword, _szPassword);
strcpy(pPWFolderInfo->szCurrentPassword, _szCurrentPassword);
strcpy(pPWFolderInfo->szUserid, _szUserid);
```

This code must be inserted in the C source file by the programmer, after the file is generated by the SOM Precompiler. This may be done using a normal text editor.

7 Finally, the SOM Precompiler provides a default zero return statement, typecast with the return data type of the method as declared in the methods section of the class definition file. This statement may be altered by the programmer if required, provided that consistency with the method's prototype and declaration is maintained.

7.4 Object Behavior

The behavior of an object in the Workplace Shell is very similar to that of a window under Presentation Manager. An object must have its class registered with the system, an instance of that class must be created ("instantiated") in the system, and that instance (and any other instance) then receives messages and uses its methods to process these messages. When processing is completed, the instance may be destroyed.

One significant difference between a Workplace Shell object class and a window class under Presentation Manager is that Workplace Shell object classes are normally **persistent**; that is, while a Presentation Manager window class is defined only for the duration of the application's execution, a Workplace Shell object class remains defined to the system, and is useable by any application until such time as it is explicitly deregistered from the system.

7.4.1 Creating an Object

A new object class in the Workplace Shell is typically created by taking an existing object class and subclassing it, introducing new data and methods, and modifying existing behaviors where required. The new object class is then registered with the Workplace Shell, and is available from that point on.

7.4.1.1 Registration

Once an object class has been defined, compiled and placed into a dynamic link library, it must be registered with Workplace Shell before it can be used. This may be accomplished in any of two ways:

- An object class may be registered with the Workplace Shell using the **WinRegisterObjectClass()** function. This function records the name of the object class, and the name of the DLL that contains the code to implement the class. Note that if specifying a fully qualified path name for *pszModName*, then the DLL does not need to be placed in the LIBPATH.
- Additionally an object may also be registered with the Workplace Shell using the **SysRegisterObjectClass()** function from REXX. Like the WinRegisterObjectClass, this function also records the name of the object class, and the name of the DLL that contains the code to implement the class.

An example of the **WinRegisterObjectClass()** function is given in Figure 47, and an example of the **SysRegisterObjectClass()** function is given in Figure 48 on page 123.

```
PSZ  pszClassName = "NewObject";                /* Class name           */
PSZ  pszModName = "NEWOBJ";                     /* DLL module for class  */
BOOL bSuccess;                                  /* Success flag         */

bSuccess = WinRegisterObjectClass(pszClassName, /* Register class       */
                                  pszModName);  /* DLL module name      */
```

Figure 47. Registering a Workplace Shell Object Class

Figure 47 provides a very simple example; a useful technique for registering object classes is to build a simple program that reads a set of strings from an ASCII data file and uses these strings as parameters to the **WinRegisterObjectClass()** function. In this way, a generic object-registration routine can be built and used for multiple object classes, without the need to modify and recompile source code.

Figure 48 on page 123 shows a sample piece of REXX code that registers a class called pwFolder to the Workplace Shell. Notice that the DLL which contains the pwFolder Workplace Object is also copied from the current directory. If this copy was unsuccessful, the author of this code assumed this was because the Workplace Shell has the DLL opened and so the REXX code deregisters the class from the Workplace Shell.

```
/*  */
Call RxFuncadd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
Call SysLoadFuncs

'@echo off'

'copy pwfolder.dll c:\os2\dll 1>nul: 2>nul:'

if rc then do
   say 'Error DLL could not be updated please re-boot'
   /* Remove bad entry */
   RetCode = SysDeregisterObjectClass( "PWFolder");
   'pause'
   exit(1)
   end

RetCode = SysRegisterObjectClass( "PWFolder", "pwfolder")

if RetCode then
   say 'PWFolder Class registered'
else do
   say 'Error PWFolder Class failed to register'
   /* Remove false entry */
   RetCode = SysDeregisterObjectClass( "PWFolder");
   exit(1)
   end
```

Figure 48. REXX Code to Register a Workplace Object

Note that once an object class has been registered with the Workplace Shell, it is permanently available until it is explicitly deleted by deregistering it. See 7.4.4, "Deregistering an Object Class" on page 147 for information on deregistering an object class.

7.4.1.2 Class Data

Class data is owned by the object class rather than by an instance of that class. It is therefore available to all instances of the class, and must be initialized prior to instantiating any objects within the class.

For this reason, class data is initialized when the object classes are loaded from their DLLs, either during Workplace Shell initialization or dynamically during execution. Class data initialization is performed by the _wpclsInitData class method, which is called by the system when the class is loaded. If a new object class has class data that must be initialized, it should override the _wpclsInitData method and perform its class-specific processing.

An example of an overridden _wpclsInitData method from the password-protected folder example is shown in Figure 49 on page 124.

```
HMODULE   hModule;
   :
   :
SOM_Scope void SOMLINK pwfoldercls_wpclsInitData(M_PWFolder *somSelf)
{
    CHAR  ErrorBuffer[100];                    /* Error buffer          */

    /* M_PWFolderData *somThis =
                        M_PWFolderGetData(somSelf); */
    M_PWFolderMethodDebug("M_PWFolder",        /* Set debug info        */
                          "pwfoldercls_wpclsInitData");

    DosLoadModule((PSZ) ErrorBuffer,           /* Get module handle     */
                  sizeof(ErrorBuffer),         /* Size of error buffer  */
                  "PWFOLDER",                  /* Name of DLL           */
                  &hModule);                   /* Module handle         */

    parent_wpclsInitData(somSelf);             /* Allow default proc    */
}
```

Figure 49. Initializing Class Data

In the example shown in Figure 49, a global variable *hModule* is used to contain the module handle for the DLL, which is required when loading Presentation Manager resources such as strings, pointers or dialogs. Since a global variable is used rather than a class data variable, the first statement in the overridden method, which obtains a handle to the class data, is not required and is therefore commented out.

Any class data items obtained or initialized by an object class from within the _wpclsInitData method should also be freed by the object class, by overriding the _wpclsUnInitData method. This method is invoked by the system when an object class is deregistered (see 7.4.4, "Deregistering an Object Class" on page 147), or when the Workplace Shell process is terminated. An example of the _wpclsUnInitData method is shown in Figure 50.

```
SOM_Scope void   SOMLINK pwfoldercls_wpclsUnInitData(M_PWFolder *somSelf)
{
    /* M_PWFolderData *somThis
                        = M_PWFolderGetData(somSelf); */
    M_PWFolderMethodDebug("M_PWFolder",        /* Set debug info        */
                          "pwfoldercls_wpclsUnInitData");

    DosFreeModule(hModule);                    /* Free module handle    */

    parent_wpclsUnInitData(somSelf);           /* Allow default proc    */
}
```

Figure 50. Freeing Class Data Items

The example shown in Figure 50 assumes that the module handle for the DLL has already been obtained and stored in the global variable *hModule*, as shown in Figure 49.

7.4.1.3 Instantiation

Once an object class has been registered with the Workplace Shell, an instance of that class may be created; this is known as **instantiation**. This may be done in one of three ways. One of the simplest method is to open the *Templates* folder and drag the template for the object class to the required location. Alternatively, an object may be created from within an application using the WinCreateObject() function. An example of this is shown in Figure 51. And lastly Figure 52 shows a sample piece of REXX code that creates a Workplace Object called pwFolder, along with some parameters for the object.

```
PSZ  pszClassName   = "NewObject";              /* Class name          */
PSZ  pszObjectTitle = "My New Object";          /* Object title        */
PSZ  pszParams = "ICON=C:\\ICONS\\MYNEWICON.ICO"; /* Setup string      */
PSZ  pszLocation = "C:\\Desktop\\MyNewFolder";  /* Location for object */

ULONG ulFlags;                                  /* Creation flags      */

HOBJECT hObject;                                /* Object handle       */

hObject = WinCreateObject(pszClassName,         /* Create object       */
                          pszObjTitle,          /* Title for icon      */
                          pszParams,            /* Setup string        */
                          pszLocation,          /* Location for object */
                          CO_REPLACEIFEXISTS);  /* Creation flags      */
```

Figure 51. C Code to Create an Object

```
/*  */
Call RxFuncadd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
Call SysLoadFuncs

'@echo off'

RetCode = SysCreateObject( "PWFolder", "FinanceFile", "<WP_DESKTOP>",
                           "PASSWORD=wps;OBJECTID=<MyFinanceFile>")

if RetCode then
   say 'PWFolder Object created'
else do
   say 'Error creating object'
   exit(1)
   end
```

Figure 52. REXX Code to Create an Object

Note that the *pszParams* parameter shown in Figure 51 is used to contain a *setup string*, which can be used to pass one or more of a number of parameters to the object class. In the example, it is used only to set the icon for the object, but may also be used to specify other parameters for that instance of the class. The keywords and values supported by the *WPObject* class are documented in the *IBM OS/2 Version 2.0 Presentation Manager Reference*; other object classes may add their own keywords and values.

The final parameter contains one or more flags which determine the behavior of the WinCreateObject() call if the object being created clashes with an object that

already exists with the specified name and in the specified location. Valid actions are for the call to fail, to update the existing object or to replace the existing object. These flags are documented in the *IBM OS/2 Version 2.0 Presentation Manager Reference*.

The setup string is passed as a parameter to the method, which can either be invoked when the object is instantiated, or during a call to WinSetObjectData. Because a call can be made to the _wpSetup method from WinSetObjectData, you must not process any default settings other than those related to the parameters passed to the _wpSetup method. This method is defined by the *WPObject* class, and may be overridden by a new object class in order to check for its own keywords and take appropriate setup action.

The _wpSetup method accepts the setup string as a parameter, and may then parse the setup string, extract any class-specific data and perform appropriate processing on that data. However, since many of the keywords that may be specified in the setup string are defined by the *WPObject* class and are handled by the default _wpSetup method, the default processing must be carried out. In this particular case, the default processing may be carried out before or after the class-specific processing.

An example of an overridden _wpSetup method is shown in Figure 53 on page 127; this example shows the use of an additional parameter in the setup string (PASSWORD=) to set an initial password for a password-protected folder upon folder creation. The setup string is parsed from within the object by calling the _wpScanSetupString method. Both of these methods, along with the keywords supported by the *WPObject* class, are described in the *IBM OS/2 Version 2.0 Presentation Manager Reference*.

After performing the class-specific processing in the _wpSetup method, an object class should invoke its parent class's _wpSetup method to perform the default processing for any other keywords in the setup string that are defined by the parent class.

Before the _wpSetup method is invoked, the system invokes the object's _wpInitData method, which allows an object to allocate resources and initialize its instance data. See 7.4.1.4, "Instance Data" on page 127 for further details.

Note that unlike a Presentation Manager window, which exists only for the duration of an application's execution, an object remains in existence permanently unless explicitly deleted from the system.

```
SOM_Scope BOOL SOMLINK pwfolder_wpSetup(PWFolder *somSelf,
                                        PSZ pszSetupString)
{
    CHAR  pszInitPword[20];                    /* Buffer for password   */
    BOOL  bFound;                              /* Success flag          */
    ULONG ulRetLength;

    PWFolderData *somThis =                    /* Get instance data     */
                PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",            /* Set debug info        */
                        "pwfolder_wpSetup");

    if (pszSetupString != NULL)                /* If string is present  */
       {
       bFound=_wpScanSetupString(somSelf,       /* Scan setup string to */
                                 pszSetupString, /* find keyword        */
                                 "PASSWORD"
                                 pszInitPword,
                                 &RetLength);

       if (bFound)                             /* If parameter present  */
          {
          strcpy(_szPassword,                  /* Copy p'word to folder */
                 pszInitPword);                /* p'word and current    */
          strcpy(_szCurrentPassword,           /* p'word - initialize   */
                 pszInitPword);                /* in unlocked state     */
          }
       }
    return (parent_wpSetup(somSelf,            /* Allow default proc to */
                           pszSetupString));   /* occur                 */
}
```

Figure 53. Object Setup. This example shows an overridden _wpSetup method which parses the setup string to extract class-specific parameters.

7.4.1.4 Instance Data

When an object is created or awakened from a dormant state, the _wpInitData method is invoked by the system. This method allows an object to initialize its instance data to a known state. Operating system resources should be allocated at this stage, but Presentation Manager resources should not, since a view of the object is not yet being opened. The allocation of Presentation Manager resources is typically done during processing of the _wpOpen method (see 7.4.2.1, "Opening an Object" on page 128).

If an object has its own instance data, which must be initialized to a known state before processing may be carried out, the object should override the _wpInitData method in its class definition file, and include the initialization code. However, for any object class other than a base storage class, the default initialization processing must be carried out in addition to the class-specific processing. This allows the correct initialization of any instance data items defined by the parent class, and ensures that the new object class behaves in a manner consistent with its ancestors.

Figure 54 on page 128 shows an overridden _wpInitData method, which initializes the password information for a password-protected folder.

```
SOM_Scope void SOMLINK pwfolder_wpInitData(PWFolder *somSelf)
{
    CHAR  ErrorBuffer[100];                    /* Error data buffer     */

    PWFolderData *somThis =                    /* Get instance data     */
                PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",            /* Set debug info        */
                        "pwfolder_wpInitData");

    strcpy(_szCurrentPassword,                 /* Initialize folder     */
           "password");                        /* password              */
    strcpy(_szPassword,                        /* Set current password  */
           "password");                        /* to folder password    */
                                               /* ie. Set unlocked      */

    return(parent_wpInitData(somSelf));        /* Perform default proc  */
}
```

Figure 54. Initializing Instance Data

Note that during processing of the _wpInitData method, the instance data of the object is not necessarily in a known state. The programmer must therefore take great care when carrying out any processing during the execution of this method, in order to avoid using data that may not yet have been initialized correctly. Failure to follow this guideline may cause unpredictable results for the object.

7.4.2 Using an Object

A user typically accesses an object by opening a **view** of that object. For example, to access the contents of a folder object, the user opens the default view (usually an icon view) of the folder, which then displays its contents. This is certainly true for container objects such as folders, and for the password-protected folder class used as an example in this chapter, although other "device" objects such as printers or the shredder may be used without a view.

When no view of an object is open, *and* the folder within which the object resides is not open, the object is said to be **dormant**; typically, no system resources are allocated to the object and its instance data is in an unknown state. Opening and closing views of an object therefore involve not only the opening and closing of windows, but also allocating and freeing resources, and saving and restoring the instance data of the object. Similarly, opening a folder requires saving and restoring the instance data of the objects in that folder.

7.4.2.1 Opening an Object

As mentioned above, a user typically interacts with an object using a view of that object. An object may support various types of view; for example, the *WPFolder* object class supports icon, tree, details and settings views. By default, an object class supports the view types defined by its ancestors, and a programmer may also define new view types for the object class.

When a view of an object is opened, the _wpViewObject method is invoked by the Workplace Shell. This method determines if there is already an open view of the view specified for the object. If there is not then _wpOpen is called to open a view for the object. If there already is an open view, _wpViewObject checks the

objects settings for concurrent views. If concurrent views are set, the _wpOpen is called to open an additional view of the object with the specified view. Therefore your programs should call _wpViewObject and not _wpOpen, but override _wpOpen to add your own unique views. Note that the concurrent view setting can normally be found on an object's settings notebook, under the "Window" tab, and is labelled "Object Open Behavior".

The _wpOpen method is defined and implemented by the base storage class *WPObject*, and may be overridden by a new object class to perform its own class-specific processing. The supported views for each object class are implemented as part of the _wpOpen method, using Presentation Manager windows.

When a view is opened by the user from a context menu, the _wpMenuItemSelected method is invoked (see 7.2.1.6, "Attaching a Method to the Context Menu" on page 108 for more detailed discussion of this method). The _wpMenuItemSelected method typically invokes the _wpViewObject method, which may invoke the _wpOpen method as outlined above.

When the user opens a view by double-clicking the mouse on an object's icon, the _wpViewObject method invokes the _wpOpen method and passes an OPEN_DEFAULT value. The default processing for the _wpOpen method invokes the _wpQueryDefaultView method to determine the default view for the object, and immediately invokes the _wpOpen method a second time with the identifier for that view.

An example of an overridden _wpOpen method is given in Figure 55 on page 130. This example shows a password-protection facility being added to a folder to prevent access by unauthorized users. Upon invocation of the _wpOpen method, the password-protected folder object class displays a dialog box to accept a password from the user. It then compares that password with the correct password for that folder before actually opening the folder. Visual cues such as the folder's icon and the word "Locked" on the folder's title are modified or removed during the _wpOpen processing.

```
SOM_Scope HWND SOMLINK pwfolder_wpOpen(PWFolder *somSelf,
                                       HWND hwndCnr,
                                       ULONG ulView,
                                       ULONG param)
{   ULONG ulResult;
    CHAR  szTitle[100];

    PWFolderData *somThis =
                PWFolderGetData(somSelf);       /* Set instance data     */
    PWFolderMethodDebug("PWFolder",             /* Set debug info        */
                        "pwfolder_wpOpen");

    if ((strcmp(_szCurrentPassword,             /* If not locked         */
                _szPassword)) == 0)
       return(parent_wpOpen(somSelf,            /* Allow open to proceed */
                            hwndCnr,            /* in normal way, using  */
                            ulView,             /* default processing    */
                            param));

    ulResult = WinDlgBox(HWND_DESKTOP,          /* Display p'word dialog */
                         HWND_DESKTOP,          /* Desktop is owner      */
                         dpPassword,            /* Dialog procedure      */
                         hModule,               /* Module handle         */
                         DLG_PASSWORD,          /* Dialog resource id    */
                         (PVOID)somSelf);       /* Object pointer        */

    if (ulResult == DID_OK)                     /* If not cancelled      */
       {
       if ((strcmp(_szCurrentPassword,          /* If correct password   */
                   _szPassword)) == 0)
          {
          strcpy(szTitle,                       /* Get title string      */
                 _wpQueryTitle(somSelf));
          szTitle[strlen(szTitle)-9] = '\0';    /* Remove <LOCKED>       */
          _wpSetTitle(somSelf,szTitle);         /* Reset title string    */

          <Set icon to unlocked state>

          return (parent_wpOpen(somSelf,        /* Allow default _wpOpen */
                                hwndCnr,        /* processing to occur   */
                                ulView,         /* by invoking parent's  */
                                param));        /* method                */
          }
       else
          {
          WinMessageBox(HWND_DESKTOP,           /* Display message box   */
             HWND_DESKTOP,
             "Password incorrect. Folder remains locked.",
             "Password Failed",
             0, MB_OK | MB_CUAWARNING);
          return((BOOL)0);                      /* Return FALSE          */
          }
       }
}
```

Figure 55. Opening an Object. This example shows the _wpOpen method, which is called by the system when a view of an object is opened, being overridden to add password protection to a folder.

Since the view being opened in this case is a view defined by the *WPFolder* class, the actual opening of the view and presentation of the folder's contents is handled using the default processing supplied by the parent class, which is called after the class-specific processing has completed.

If an object class wishes to create a new view, it must add the name of the view to the *Open* submenu in the object's context menu, and include a case for that view in the _wpMenuItemSelected method. This method then invokes _wpViewObject with a specific value in the *ulView* parameter, indicating the view to be opened. The class-specific processing for _wpOpen must test for this value, open a window and display the correct information using Presentation Manager functions.

The example in Figure 55 does not include the code to set the folder's icon to the "unlocked" state. This code is identical to the code used in Figure 40 on page 108 to set the icon to the "locked" state; the resource identifier of the "unlocked" icon is simply substituted in the _wpOpen method for the identifier of the "locked" icon.

Note that in many cases, it is important for an object class to allow the default processing for _wpOpen to occur *before* it attempts to carry out its own processing. This allows instance data and control information to be established and initialized before the object attempts any processing using these items. In Figure 55 on page 130 however, the additional class-specific processing determines whether the object should open *at all*; if processing is allowed to proceed, no alteration to the default processing takes place. The default processing may therefore be carried out after the additional class-specific processing introduced by the password-protected folder class.

The default processing for the _wpOpen method supports a number of views, depending upon the parent class of the object; for example, the processing for the *WPFolder* class supports ICON, TREE and DETAILS views. For new object classes which support additional views, the _wpOpen method must be overridden and the additional view types opened explicitly as windows using appropriate Presentation Manager functions. Since a view of an object is essentially a window, new views can be implemented as normal Presentation Manager windows and the correct information displayed using text or graphical programming functions, according to the requirements of the object class.

The application must always define a new view if it introduces one. Never process OPEN_DEFAULT other than passing it to the parent class. If you want to have your own view be the default, then override the _wpclsQueryDefaultView method.

Note that upon opening a view using a Presentation Manager window, an object should add itself to the "Use List" maintained by the Workplace Shell. If the view is the first view of the object to be opened, this causes the Workplace Shell to modify the object's icon to indicate the "in use" state. The object should also register the view with the Workplace Shell, which will then subclass the view's frame window, automatically attach the object's context menu to the window's system menu icon, and add the view to the Workplace Shell's Window List. These steps are done using the _wpAddToObjUseList and _wpRegisterView methods, as shown in Figure 56 on page 132.

```
HWND    hView;                                  /* View window handle    */

typedef struct _OBJECTVIEW                      /* Object view structure */
               {
               SOMAny   *Object;                /* Object pointer        */
               USEITEM  UseItem;                /* USEITEM structure     */
               VIEWITEM ViewItem;               /* VIEWITEM structure    */
               } OBJECTVIEW;

OBJECTVIEW *pObjectView;                        /* Pointer to structure  */

<Create Window>                                 /* Get window handle     */

pObjectView = _wpAllocMem(somSelf,              /* Allocate memory       */
                          sizeof(OBJECTVIEW),   /* Size of mem object    */
                          NULL);
pObjectView->Record          = somSelf;         /* Initialize OBJECTVIEW */
pObjectView->UseItem.type    = USAGE_OPENVIEW;  /* structure             */
pObjectView->ViewItem.view   = OPEN_CUST;
pObjectView->ViewItem.handle = hView;

WinSetWindowULong(hView,                        /* Store pointer to      */
                  QWL_USER,                     /* structure in window   */
                  (ULONG)pObjectView);          /* words                 */

_wpAddToObjUseList(somSelf,                     /* Add to Use List       */
                   &pObjectView->UseItem);      /* USEITEM structure     */
_wpRegisterView(somSelf,                        /* Register view         */
                hView,                          /* View window handle    */
                "Customer Details")             /* Title of view         */
```

Figure 56. Opening a Custom View of an Object

The Workplace Shell makes use of a USEITEM and a VIEWITEM structure in the _wpAddToObjUseList method. It assumes that these structures are contiguous in memory; hence they should be allocated as part of a larger data structure such as the OBJECTVIEW structure shown in Figure 56. A pointer to this structure is stored in the window words of the view window, so that information such as the object's pointer can be accessed from the view's window procedure.

Note that upon closing a view, the view's window procedure should invoke the _wpDeleteFromObjUseList method to remove the view from the Use List. If the view is the only open view of the object, the object's icon is modified to remove the "in use" emphasis.

7.4.2.2 A Custom View of our pwFinanceFile

This section shows how to implement a custom view of the pwFinanceFile Workplace Object. Not all of the code, such as the resource files, header files, are shown. For a full code listing please refer to Appendix E, "Source Code for the PWFolder and PWFinanceFile objects" on page 347

Figure 57 on page 133 shows how the "Open Finance File" menu item is added onto the "Open" menu item, which appears on the pwFinanceFile's context menu. Note that we also add the "Lock Finance File" menu item. Figure 58 on page 134 shows the resulting context menu which is provided by Figure 57 on page 133. Note that the "OS/2 System Editor" menu item is also present. The

Workplace Shell wpDataFile class has added this due to file type associations. In this case because this object does not have a file type, it is assumed by the ancestor classes to be text and the default association for the text type is the OS/2 System Editor. We could remove this by overriding the _wpclsQueryDefaultView method, thus making our view the default one.

```
/*
 *
 *   METHOD:   wpModifyPopupMenu                          PUBLIC
 *
 *   PURPOSE:  Adds an additional "Lock" item to the object's context menu.
 *             Adds a "Open Finance File" item to the "Open" item
 *   INVOKED:  By Workplace Shell, upon instantiation of the object instance.
 *
 */

SOM_Scope BOOL   SOMLINK pwFinanceFile_wpModifyPopupMenu(PWFinanceFile *somSelf,
               HWND hwndMenu,
               HWND hwndCnr,
               ULONG iPosition)
{
    PWFinanceFileData *somThis =                      /* Get instance data pointer */
                PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile",             /* Set debug info          */
                        "pwFinanceFile_wpModifyPopupMenu");

    _wpInsertPopupMenuItems(somSelf,            /* Insert menu item         */
                            hwndMenu,           /* Menu handle              */
                            iPosition,          /* Default position         */
                            hmodThisClass,      /* Module handle            */
                            ID_CXTMENU_LOCK,    /* Menu item identifier     */
                            0);                 /* No submenu identifier    */

    _wpInsertPopupMenuItems( somSelf,           /* Insert menu item         */
                             hwndMenu,          /* Menu handle              */
                             0,                 /* at the top!              */
                             hmodThisClass,     /* Module handle            */
                             ID_OPENFinanceFile, /* Menu item identifier    */
                             WPMENUID_OPEN);    /* Submenu identifier       */


    return(parent_wpModifyPopupMenu(somSelf,    /* Invoke default processing */
                                    hwndMenu,
                                    hwndCnr,
                                    iPosition));
}
```

Figure 57. _wpModifyPopupMenu .C code. This example shows how to add two menu items to the "Open Finance File" menu item on the pwFinanceFile's context menu.

Figure 60 on page 135, Figure 61 on page 136, Figure 62 on page 138 and Figure 63 on page 141 show the additional code required to process the selection of the "Open Finance File" menu item. Figure 59 on page 134 shows the window view which is presented when the user selects the "Open Finance File" menu item. This window is empty and the population of the window with information from the file associated with the instance of the pwFinanceFile is a normal PM programming exercise which is left for the reader to perform.

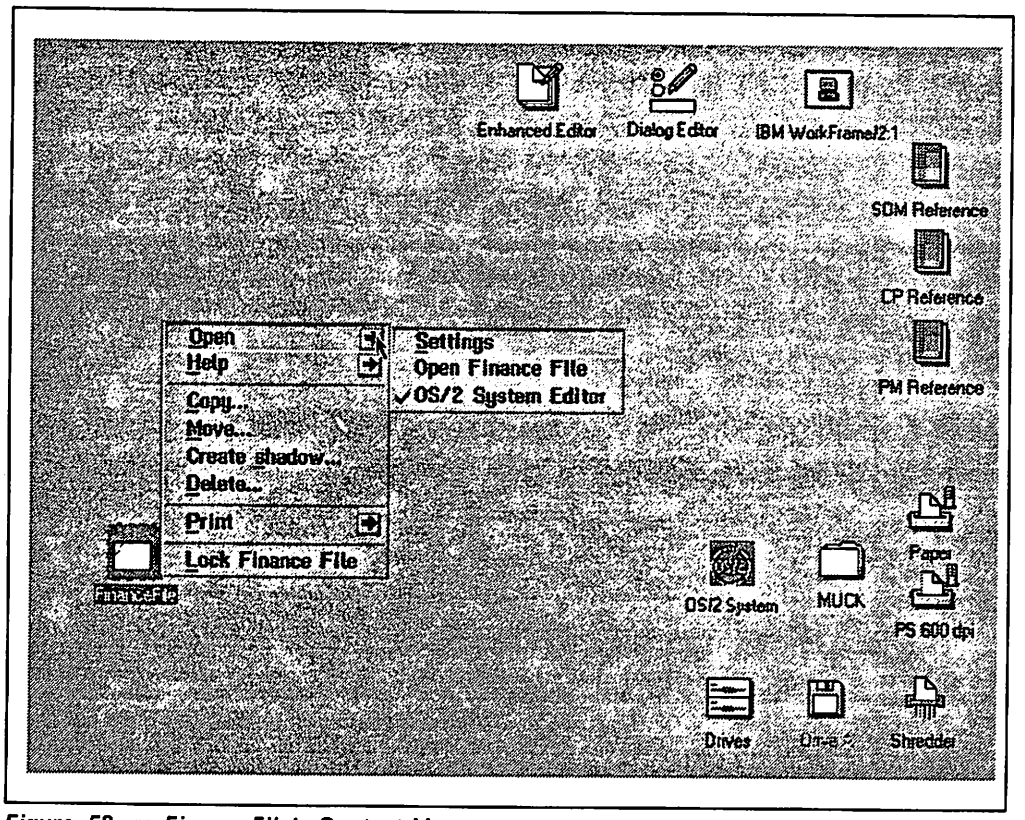

Figure 58. pwFinanceFile's Context Menu

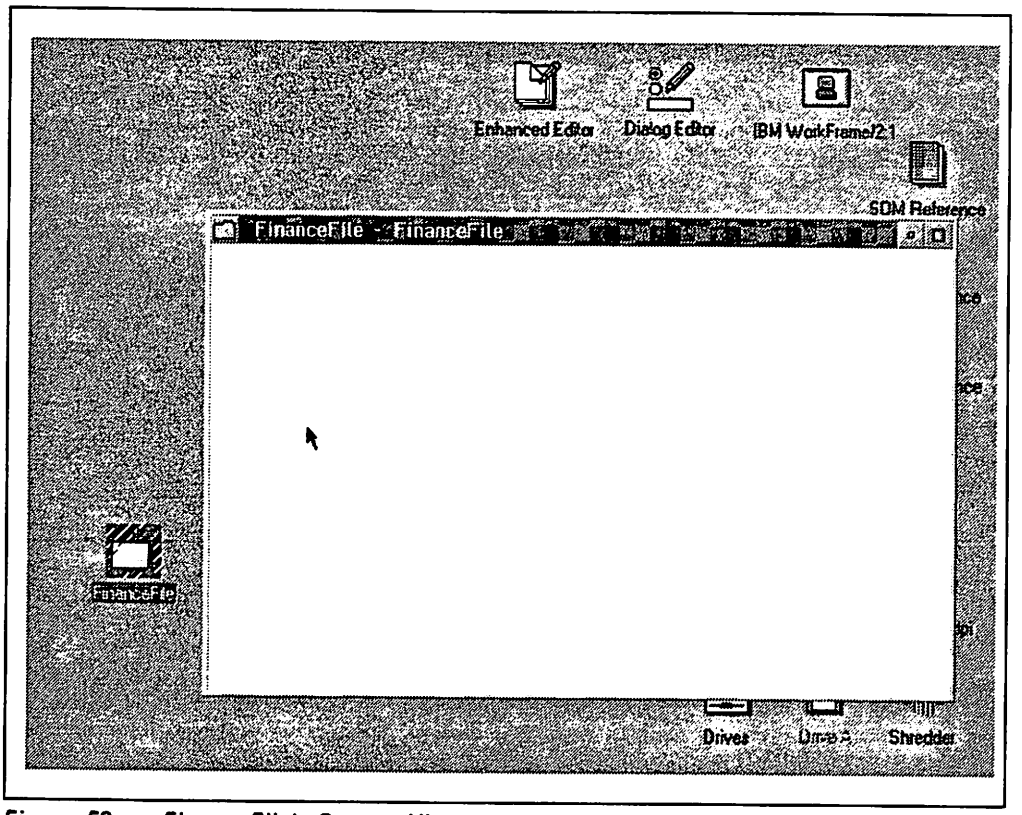

Figure 59. pwFinanceFile's Custom View

```
/*
 *
 *   METHOD:   wpMenuItemSelected                          PUBLIC
 *
 *   PURPOSE:  Processes the user's selections from the context menu.  The
 *             overridden method processes the added "Lock" & "OPENFinanceFile"
 *             items, and passes all others to the parent method
 *
 *   INVOKED:  By Workplace Shell, upon selection of a menu item by the user.
 *
 */

SOM_Scope BOOL   SOMLINK pwFinanceFile_wpMenuItemSelected(PWFinanceFile *somSelf,
                 HWND hwndFrame,
                 ULONG ulMenuId)
{
    PWFinanceFileData *somThis =                      /* Get instance data pointer */
                  PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile",         /* Set debug info            */
                        "pwFinanceFile_wpMenuItemSelected");

    switch( ulMenuId )                          /* Switch on item identifier     */
    {
       case IDM_LOCK:                           /* Lock item selected            */
          _LockFinanceFile(somSelf);            /* Invoke _LockFinanceFile method */
          break;

    /*
     *   We could call wpOpen here, but if the object is already opened
     *   the following API determines whether the object should be
     *   resurfaced or if multiple views are desired.
     *   Must call wpViewObject not wpOpen.  If you use wpOpen then multiple
     *   concurrent views won't work.  User can set object to open multiple views
     *   or switch to.
     */

       case IDM_OPENFinanceFile:                /* Open a view selected      */
          _wpViewObject(somSelf, NULLHANDLE, OPEN_FinanceFile, 0);
          break;


       default:                                 /* All other items           */
          parent_wpMenuItemSelected(somSelf,    /* Invoke default processing */
                                    hwndFrame,
                                    ulMenuId);
          break;
    }
}
```

Figure 60. _wpMenuItemSelected .C code

"C" Code

```
/*
 *
 *   METHOD:   wpOpen                                        PUBLIC
 *
 *   PURPOSE:  Only allows a FinanceFile to be opened if the FinanceFile is unlocked, or
 *             if the user supplies the correct password in response to the
 *             dialog.
 *
 *   INVOKED:  By Workplace Shell, upon selection of the "Open" menu item by
 *             the user.
 *
 */

SOM_Scope HWND   SOMLINK pwFinanceFile_wpOpen(PWFinanceFile *somSelf,
                 HWND hwndCnr,
                 ULONG ulView,
                 ULONG param)
{
    ULONG    ulResult;                          /* Return value              */
    CHAR     szTitle[100];                      /* FinanceFile title buffer      */
    PVOID    pCreateParam;
    BOOL     bAllowAccess = FALSE;              /* user is allowed in */

    PWFinanceFileData *somThis =                     /* Get instance data pointer */
                 PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile",            /* Set debug info           */
                        "pwFinanceFile_wpOpen");

    if ((strcmp(_szCurrentPassword,             /* If FinanceFile is locked   */
        _szPassword)) != 0)
    {
       somPrintf("ask for a password\n");
    pCreateParam = malloc( sizeof(ULONG) );     /* Allocate memory to pass a */
                                                /* ULONG to the dialog proc  */
    *((PULONG)pCreateParam) = (ULONG)somSelf;   /* Put the somSelf pointer   */
                                                /* in the CreateParam memory */

    ulResult = WinDlgBox(HWND_DESKTOP,          /* Display password dialog   */
                         HWND_DESKTOP,          /* Desktop is owner          */
                         PasswordDlgProc,       /* Dialog procedure address  */
                         hmodThisClass,         /* Module handle             */
                         ID_DLG_PASSWORD,       /* Dialog resource id        */
                         pCreateParam );        /* Create Param holding the  */
                                                /* pointer to this object    */
```

Figure 61 (Part 1 of 2). _wpOpen

```
    if (ulResult == DID_OK )                    /* If user hit OK button     */
       {
       if ((strcmp(_szCurrentPassword,          /* If password is correct    */
                 _szPassword)) == 0)
          {
          strcpy(szTitle,                       /* Get title string          */
                 _wpQueryTitle(somSelf));
          szTitle[strlen(szTitle)-9] = '\0';    /* Remove <LOCKED>           */
          _wpSetTitle(somSelf,szTitle);         /* Reset title string        */

          _wpSetIcon(somSelf,                   /* Set icon to unlocked      */
                     hUnlockedIcon);            /* state                     */

          /* now we can allow the user access to the object proper ! */
          bAllowAccess = TRUE;

          }
       else                                     /* Password is incorrect     */
          {
          WinMessageBox(HWND_DESKTOP,           /* Display message to user   */
                        HWND_DESKTOP,
                        "Password incorrect. FinanceFile remains locked.",
                        "Password Failed",
                        0,
                        MB_OK |
                        MB_CUAWARNING );
          return((HWND)0);                      /* Return NULL handle        */
          }
       }
    } else {
      bAllowAccess = TRUE;
    }
    if (bAllowAccess) {
       switch (ulView) {
       case OPEN_FinanceFile:
         if (!_wpSwitchTo(somSelf, ulView)) {
           /* Create a basic frame and client window for this instance */
           return PWFinanceFileInit(somSelf);
         } /* endif */
         break;

       default:
         return(parent_wpOpen(somSelf,          /* Allow open to proceed in  */
                              hwndCnr,          /* normal way using default  */
                              ulView,           /* processing                */
                              param));
       } /* endswitch */
    } else {
    } /* endif */
}
```

Figure 61 (Part 2 of 2). _wpOpen

```
/*************************************************************************
*                                                                        *
*       ROUTINE:    PWFinanceFileInit()                                  *
*                                                                        *
*       DESCRIPTION:  PWFinanceFile Inisialisation                       *
*                                                                        *
*       RETURNS:    Handle of PWFinanceFile frame window, NULL if error  *
*                                                                        *
*************************************************************************/
HWND PWFinanceFileInit (PWFinanceFile* somSelf)
{

   HAB  hab;                                        /* PM anchor block handle */
   HWND hwndFrame = NULLHANDLE;                        /* Frame window handle */
   HWND hwndClient = NULLHANDLE;
   PWINDOWDATA pWindowData;
   BOOL fSuccess;
   SWCNTRL    swcEntry;                                        /* Switch Entry */
   FRAMECDATA flFrameCtlData;                                /* Frame Ctl Data */

   somPrintf("PWFinanceFileInit\n");

   hab = WinQueryAnchorBlock(HWND_DESKTOP);
   if (!WinRegisterClass( hab , szFinanceFileWindowClass, (PFNWP)FinanceFileWndProc ,
                          CS_SIZEREDRAW | CS_SYNCPAINT, sizeof(pWindowData)))
   {
      somPrintf("FinanceFileInit Failure in WinRegisterClass\n");
      return NULLHANDLE ;
   }

   /*
    *   Allocate some instance specific data in Window words of Frame window.
    *   This will ensure our window procedure can use this object's methods
    *   (our window proc isn't passed a * somSelf pointer).
    */
   pWindowData = (PWINDOWDATA) _wpAllocMem(somSelf, sizeof(*pWindowData), NULL);

   if (!pWindowData)
   {
      somPrintf("FinanceFileInit wpAllocMem failed to allocate pWindowData\n");
      return NULLHANDLE;
   }

   memset((PVOID) pWindowData, 0, sizeof(*pWindowData));
   pWindowData->cb = sizeof(*pWindowData);             /* first field = size */
   pWindowData->somSelf = somSelf;

   /* Create a frame window
    */
   flFrameCtlData.cb            = sizeof( flFrameCtlData );
   flFrameCtlData.flCreateFlags = FCF_SIZEBORDER | FCF_TITLEBAR | FCF_SYSMENU |
                                  FCF_MINMAX ;
   flFrameCtlData.hmodResources = hmodThisClass;
   flFrameCtlData.idResources   = ID_UNLOCK;
```

Figure 62 (Part 1 of 3). pwFinanceFile's Initialization Function

```
hwndFrame =                                          /* create frame window */
WinCreateWindow(
   HWND_DESKTOP,                /* parent-window handle                    */
   WC_FRAME,                    /* pointer to registered class name        */
   _wpQueryTitle(somSelf),      /* pointer to window text                  */
   0,                           /* window style                            */
   0, 0, 0, 0,                  /* position of window                      */
   NULLHANDLE,                  /* owner-window handle                     */
   HWND_TOP,                    /* handle to sibling window                */
   (USHORT) ID_FRAME,           /* window identifier                       */
   (PVOID) &flFrameCtlData,     /* pointer to buffer                       */
   NULL);      ;                /* pointer to structure with pres. params. */

if (!hwndFrame)
{
   somPrintf("FinanceFileInit Failure in WinCreateWindow\n");
   return NULLHANDLE;
}
hwndClient =          /* use WinCreateWindow so we can pass pres params */
WinCreateWindow(
   hwndFrame,                   /* parent-window handle                    */
   szFinanceFileWindowClass,    /* pointer to registered class name        */
   NULL,                        /* pointer to window text                  */
   0,                           /* window style                            */
   0, 0, 0, 0,                  /* position of window                      */
   hwndFrame,                   /* owner-window handle                     */
   HWND_TOP,                    /* handle to sibling window                */
   (USHORT)FID_CLIENT,          /* window identifier                       */
   pWindowData,                 /* pointer to buffer                       */
   NULL);                       /* pointer to structure with pres. params. */

if (!hwndClient)
{
   WinDestroyWindow(hwndFrame);
   return NULLHANDLE;
}

WinSendMsg(hwndFrame,WM_SETICON,MPFROMP(_wpQueryIcon(somSelf)),NULL);
WinSetWindowText(WinWindowFromID(hwndFrame,(USHORT)FID_TITLEBAR),
                                                    _wpQueryTitle(somSelf));

/*
 * Restore the Window Position
 */
fSuccess =
WinRestoreWindowPos(
   szFinanceFileClassTitle,                                  /* class title */
   _wpQueryTitle(somSelf),                             /* object title */
   hwndFrame);
```

Figure 62 (Part 2 of 3). pwFinanceFile's Initialization Function

```
    if (!fSuccess)
    {
     SWP        swp;

       /* Get the dimensions and the shell's suggested
        * location for the window
        */
       WinQueryTaskSizePos(hab,0,&swp);

       /* Set the frame window position
        */
       swp.fl              = SWP_SIZE|SWP_MOVE|SWP_RESTORE|SWP_ZORDER;
       WinSetWindowPos(hwndFrame, HWND_TOP, swp.x, swp.y, swp.cx,
                       swp.cy, swp.fl);
    }

    WinShowWindow(hwndFrame,TRUE);

    return hwndFrame;                                          /* success */

 }  /* end FinanceFileInit() */
```

Figure 62 (Part 3 of 3). pwFinanceFile's Initialization Function

```
/*****************************************************************************
*
*   FinanceFileWndProc()
*
*   DESCRIPTION: FinanceFile Window Procedure
*
*****************************************************************************/
MRESULT EXPENTRY FinanceFileWndProc( HWND hwnd, ULONG msg, MPARAM mp1, MPARAM mp2 )
{
  ULONG         MenuId;
  PWINDOWDATA   pWindowData;
  HWND          hwndFrame;
  CHAR          acBuffer[10];
  BOOL          fSuccess;
  CHAR          szPath[CCHMAXPATH];                                    //jt 55384
  ULONG         cbPath = CCHMAXPATH;                                   //jt 55384


   hwndFrame = WinQueryWindow(hwnd, QW_PARENT);

   switch( msg )
   {
      case WM_CREATE:

         pWindowData = (PWINDOWDATA) mp1;

         if (pWindowData == NULL)
         {
            somPrintf("FinanceFileWndProc:WM_CREATE couldn't get window words");
            return FALSE;
         }
         /*
          *   Fill in the class view/usage details and window specific data
          *   for this instance.
          */

         /*
          * Must create UseItem, add it to the object's use list and register the view
          */
         pWindowData->UseItem.type     = USAGE_OPENVIEW;
         pWindowData->ViewItem.view    = OPEN_FinanceFile;
         /*
          * Must be frame.  Be careful because this procedure is for the client.
          * Must get parent and pass that as ViewItem handle.
          */
         pWindowData->ViewItem.handle = hwndFrame;
         pWindowData->x               = 10;
         pWindowData->y               = 10;
         pWindowData->xDir            = 0;
         pWindowData->yDir            = 0;

         /*
          *   Set window pointer with object pointer and instance view info.
          *   Then add view to the in-use list so wpSwitchTo works.
          */
         WinSetWindowPtr(hwnd, QWL_USER, pWindowData);
```

Figure 63 (Part 1 of 3). pwFinanceFile's Window Procedure, FinanceFileProc()

```
        /*
         * _wpAddToObjUseList will tell the shell to store the view in
         * the internal linked list for the object to enable wpSwitchTo and other
         * methods to find the view.  The shell will also subclass the view window
         * this gives you title bar context menu when you call wpRegisterView.
         * wpRegisterView also puts the view in the window list and sets up
         * the title bar like:  "Object Title - View Title"
         */

        _wpAddToObjUseList(pWindowData->somSelf,&pWindowData->UseItem);
        _wpRegisterView(pWindowData->somSelf, hwndFrame,
                                                _wpQueryTitle(pWindowData->somSelf));
        WinSetFocus( HWND_DESKTOP, hwndFrame);

        /* what is the filename of the file */
        if (_wpQueryRealName(pWindowData->somSelf,szPath,&cbPath,TRUE))
           {
              somPrintf("File name is %s, size %i \n",szPath, cbPath);
           } else {
              somPrintf("Failed to get filename\n");
           } /* endif */

        break;

      case WM_COMMAND:

        break;

      case WM_PAINT:
         pWindowData = (PWINDOWDATA) WinQueryWindowPtr(hwnd, QWL_USER);

         if (pWindowData == NULL)
         {
            somPrintf("FinanceFileWndProc:WM_PAINT couldn't get window words\n");
            return FALSE;
         }
         else
         {
          HPS    hps;
          RECTL  rectl;

           hps = WinBeginPaint( hwnd, (HPS)NULLHANDLE, &rectl);
           WinFillRect( hps, &rectl, SYSCLR_WINDOW);
           WinEndPaint( hps );
         }
         break;
```

Figure 63 (Part 2 of 3). pwFinanceFile's Window Procedure, FinanceFileProc()

```
      case WM_CLOSE:
         {
          HAB hab;

            hab = WinQueryAnchorBlock(HWND_DESKTOP);

            pWindowData = (PWINDOWDATA) WinQueryWindowPtr(hwnd, QWL_USER);

            if (pWindowData == NULL)
            {
               somPrintf("FinanceFileWndProc:WM_CLOSE couldn't get window words\n");
               return FALSE;
            }
            fSuccess =
            WinStoreWindowPos(szFinanceFileClassTitle,_wpQueryTitle(pWindowData->somSelf),
                                                                        hwndFrame);
            /*
             * Must remove from the object UseList when window is closed.  (can be done
             * on WM_DESTROY instead)
             */
            _wpDeleteFromObjUseList(pWindowData->somSelf,&pWindowData->UseItem);
            _wpFreeMem(pWindowData->somSelf,(PBYTE)pWindowData);

            WinDestroyWindow ( hwndFrame ) ;
         }
         break;

      default:
         return WinDefWindowProc( hwnd, msg, mp1, mp2 );
   }
   return FALSE;

}   /* end FinanceFileWndProc() */
```

Figure 63 (Part 3 of 3). pwFinanceFile's Window Procedure, FinanceFileProc()

7.4.2.3 Automatic Opening Upon Instantiation

In many cases, it is desirable to automatically open a view of an object when the object is created. This may be achieved by using the OPEN= keyword in the setup string passed to the WinCreateObject() function. An example of this technique is shown in Figure 64.

```
PSZ  pszClassName   = "NewObject";              /* Class name          */
PSZ  pszObjectTitle = "My New Object";          /* Object title        */
PSZ  pszParams = "OPEN=ICON";                   /* Setup string        */
PSZ  pszLocation = "C:\\Desktop\\MyNewFolder";  /* Location for object */

ULONG ulFlags = CO_UPDATEIFEXISTS;              /* Creation flags      */

HOBJECT hObject;                                /* Object handle       */

hObject = WinCreateObject(pszClassName,         /* Create object       */
                          pszObjTitle,          /* Title for icon      */
                          pszParams,            /* Setup string        */
                          pszLocation,          /* Location for object */
                          ulFlags);             /* Creation flags      */
```

Figure 64. Automatically Instantiating an Object. This example shows the use of the OPEN= keyword to automatically open a view of an object upon creating the object.

The opening of the view specified in the OPEN= keyword is handled by the default processing for the _wpSetup method, as defined by the *WPObject* class. The default processing supports the icon, tree and details views, specified using

the ICON, TREE and DETAILS values for the OPEN= keyword respectively. For new object classes that support additional views, the _wpSetup method must be overridden and the additional view types opened explicitly as windows using appropriate Presentation Manager functions.

7.4.2.4 Closing an Object

When all open views of an object are to be closed, the _wpClose method is invoked. This method is normally invoked when the user selects the *Close* option from a view's context menu.

The _wpClose method may be overridden to perform class-specific processing for closing views, or to free system resources allocated during processing of the _wpOpen method. For example, Figure 65 shows the _wpClose method being overridden to automatically lock a password-protected folder whenever it is closed by the user.

```
SOM_Scope BOOL SOMLINK pwfolder_wpClose(PWFolder *somSelf)
{
    PWFolderData *somThis =                    /* Get instance data     */
                PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",            /* Set debug info        */
                        "pwfolder_wpInitData");
    _LockFolder(somSelf);                      /* Lock folder           */

    return(parent_wpClose(somSelf));           /* Allow default proc    */
}
```

Figure 65. Closing an Object. This example shows the _wpClose method being overridden in order to provide class-specific processing for the password-protected folder.

When a view of an object is closed, the system sends a WM_DESTROY message to the view's frame window. This allows the object to release any allocated resources and save its instance data, so that the object may be reopened in its current state at some future time.

Note that since the _wpClose method is defined by the parent class and is overridden, the default processing performed by the parent is called after the class-specific processing has completed.

7.4.2.5 Saving and Restoring the Object State

As already mentioned, an object is persistent; that is, it remains in existence even when all views of the object are closed. In order to maintain its instance data so that it may subsequently be opened in the same state in which it was closed, the object must save this data when its views are closed and restore it when a view is opened. The Workplace Shell provides methods that handle the saving and restoration of instance data on behalf of object classes; these methods are automatically invoked by the system at the appropriate times, and are described below.

When an object is made dormant, the system invokes the object's _wpSaveState method, which allows the object to save its instance data. A number of predefined methods are available to the object to save its data, such as _wpSaveString. These methods may be called by the object during the processing of its _wpSaveState method, in order to save instance data. An

example of the _wpSaveState method for the password-protected folder example is given in Figure 66 on page 145.

```
SOM_Scope BOOL SOMLINK pwfolder_wpSaveState(PWFolder *somSelf)
{
    PWFolderData *somThis =                    /* Get instance data     */
                 PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",            /* Set debug info        */
                        "pwfolder_wpSaveState");

    _wpSaveString(somSelf,                     /* Save folder password  */
                  "PWFolder",                  /* Class name            */
                   1L,                         /* Class-defined key     */
                   _szPassword);               /* String to be saved    */
    _wpSaveString(somSelf,                     /* Save current password */
                  "PWFolder",                  /* Class name            */
                  2L,                          /* Class-defined key     */
                  _szCurrentPassword);         /* String to be saved    */

    return (parent_wpSaveState(somSelf));      /* Invoke default proc   */
}
```

Figure 66. Saving an Object's State

An object's instance data items are saved in different locations, depending upon the class of the object. Object classes that are descendants of the *WPAbstract* class store their instance data in the OS/2 initialization file OS2.INI. Object classes that are descendants of the *WPFileSystem* class store their instance data in extended attributes. Since the password-protected folder class is descended from the *WPFolder* class defined by the Workplace Shell, which in turn is a descendant of the *WPFileSystem* class, the instance data of this object class is saved as extended attributes in the OS/2 file system.

The class-defined key passed to the _wpSaveString method is used to save the data item in a particular location, which can then be accessed, using the same key, to restore the item. In addition to strings, numeric data may be saved using the _wpSaveLong method, and other binary data such as application-defined data structures may be saved using the _wpSaveData method.

Note that since the _wpSaveState method is defined by the object's class's ancestors and overridden, it must invoke the default processing supplied by the parent class in order to correctly save any instance data defined by ancestor classes. Failure to do so may cause unpredictable results upon awakening the object from its dormant state.

An object must retrieve its instance data and restore its state whenever it is made awake. At this point, the system invokes an object's _wpRestoreState method, which allows the object to restore its state. During the processing of this method, the object can invoke other methods such as _wpRestoreString, which restore specific instance data items. An example of the _wpRestoreState method is given in Figure 67 on page 146.

```
SOM_Scope BOOL32   SOMLINK pwfolder_wpRestoreState(PWFolder *somSelf,
                ULONG ulReserved)
{
    ULONG ulResStrLen;                          /* String length        */

    PWFolderData *somThis =                     /* Get instance data    */
                PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",             /* Set debug info       */
                        "pwfolder_wpRestoreState");

    _wpRestoreString(somSelf,                   /* Restore folder p'word */
                     "PWFolder",                /* Class name           */
                     1L,                        /* Class-defined key    */
                     _szPassword,               /* Target string        */
                     &ulResStrLen);             /* Length restored      */
    _wpRestoreString(somSelf,                   /* Restore curr p'word  */
                     "PWFolder",                /* Class name           */
                     2L,                        /* Class-defined key    */
                     _szCurrentPassword,        /* Target string        */
                     &ulResStrLen);             /* Length restored      */

    return(parent_wpRestoreState(somSelf,       /* Invoke default proc  */
                                 ulReserved));
}
```

Figure 67. Restoring an Object's State

The class-defined key passed to the _wpRestoreString method is used to locate the required data item, and the item is restored into the specified target string variable. Numeric data can be restored using the _wpRestoreLong method, and other binary data such as application-defined structures can be restored using the _wpRestoreData method.

Since the _wpRestoreState method is an overridden method, it is important that the default processing supplied by the parent class be invoked. Failure to do so will result in any instance data defined by ancestor classes being in an unknown state, with unpredictable results.

Note that you are not restricted to the workplace methods to save and restore data. You may use any auxiliary file, OS2.INI, extended attributes or what ever means you wish.

7.4.3 Destroying an Object

A specific instance of an object class can be destroyed by the user, simply by dragging it over the Shredder object on the Workplace Shell desktop. If an object or application wishes to delete an object. it may do so using the **WinDestroyObject()** function, as shown in Figure 68.

```
HOBJECT hObject;                        /* Object handle        */
BOOL    bSuccess;                       /* Success flag         */

bSuccess = WinDestroyObject(hObject);   /* Destroy object       */
```

Figure 68. Destroying an Object

The **WinDestroyObject()** function uses the object handle that is returned by the WinCreateObject() function. The object or application that creates the object is responsible for storing this handle during the existence of the object.

When an object is destroyed, the system invokes the object's _wpUnInitData method, which may be used to free any resources or instance data items that were allocated to that particular object.

7.4.4 Deregistering an Object Class

An entire object class can be deleted from the system by deregistering it from the Workplace Shell. This is achieved by either using the **WinDeregisterObjectClass()** function, which is shown in Figure 69, or the **SysDeregisterObjectClass()** function, which is shown in Figure 70.

```
BOOL bSuccess;

bSuccess = WinDeregisterObjectClass(pszClassName);   /* Deregister class */
```

Figure 69. Deregistering an Object Class

The **WinDeregisterObjectClass()** function accepts a string containing the object class name. Once a successful call is made to the **WinDeregisterObjectClass()** function, the object class is deleted from the system and is no longer available to other objects or applications. However, the DLL that contains the code for the object class is not automatically deleted from the system; if the *Templates* folder is subsequently opened with this DLL still resident in a directory in the system's LIBPATH, a template for the class will still appear in the folder. In order to prevent this, the DLL must be explicitly deleted from the system.

During processing of the **WinDeregisterObjectClass()** call, the system invokes the object's _wpclsUnInitData method, to free any instance data or resources that were obtained when the object class was created. See 7.4.1.2, "Class Data" on page 123 for an example of this method.

Figure 70 shows a sample piece of REXX code that deregisters a Workplace Object called pwFinanceFile.

```
/*  */
Call RxFuncadd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
Call SysLoadFuncs

'@echo off'

RetCode = SysDeregisterObjectClass( "PWFinanceFile");

if RetCode then
    say 'Uninstall successfully completed for PWFinanceFile class'

say 'Re-boot NOW in order to release DLL'
'pause'
```

Figure 70. REXX Code to Deregister a WPS Object

7.4.5 Accessing Presentation Manager Resources From a Workplace Shell Object

A Workplace Shell object may access and make use of Presentation Manager resources such as icons, bitmaps, strings and dialogs. These resources may reside in the same DLL as the object's code, or in another DLL. However, since the resources must reside in a DLL, the code that loads the resources must use the **DosLoadModule** or **DosGetModuleHandle()** functions to obtain module handles, as described in 9.3.2, "Loading Resources From a DLL" on page 200. This is typically done by obtaining the module handles as part of the _wpclsInitData method, and storing them either in global variables or in class data until needed. Note that not all Workplace methods can be assumed to be called in a PM thread. You may need to create your own PM thread to access PM functions during Workplace processing.

7.5 Transient Objects

As mentioned earlier in this chapter, a Workplace Shell object differs from a Presentation Manager window in that it is persistent; that is, it continues to exist after a system restart. The exception to this rule is the **transient object**, which only exists during the current execution of the system, and is destroyed when the system is IPLed.

Transient objects are useful when a programmer wishes to represent and display information such as records from a database. As each record is retrieved, a transient object is created for that record and displayed in a folder on the Workplace Shell desktop. These objects may be opened and manipulated by the end user, but will cease to exist when the system is IPLed.

Figure 71 on page 149 shows an object window in a requester process which, upon receiving a completed database query from a server process, invokes the _wpclsNew method to create a new instance of a transient object class, representing the record retrieved from the database.

```
REPLY *Reply;                                   /* Reply data structure  */

SOMAny *NewObj;                                 /* Object pointer        */

case WMP_REQUEST_COMPLETE:
     Reply = (REPLY *)mp1;                      /* Get reply data        */

     ClassID = SOM_IdFromString("CustClass");   /* Get class SOM ID      */

     TransClass = _somFindClass(SOMClassMgrObject, /* Get class pointer  */
                                ClassID,        /* Class SOM ID          */
                                1,1);           /* Major & minor version */

     NewObj = _wpclsNew(TransClass,             /* Create new object     */
                        Reply->CustName,        /* Title for object      */
                        "",                     /* No setup string       */
                        Reply->Folder,          /* Location              */
                        TRUE);                  /* Lock flag             */

     _SetCustInfo(NewObj,                       /* Set instance data     */
                  Reply);
     break;
```

Figure 71. Creating a Transient Object

The SOM pointer to the transient object class is obtained using the **SOMIdFromString()** macro and the *_somFindClass* method (see 7.6, "Communication Between Objects" for further information). This pointer is then used to invoke the _wpclsNew method to create a new instance of the class. Once the new instance is created, a method named *_SetCustInfo*, which is defined by the transient object class, is invoked to insert the information retrieved from the database into the object's instance data.

Note that the technique shown in Figure 71 may only be used when an object is created from within the Workplace Shell process. If an object must be created from another process in the system, the WinCreateObject() function must be used.

7.6 Communication Between Objects

Objects communicate with one another in order to convey events and initiate actions on the part of other objects. Such communication is typically initiated in one of two ways:

- By an object to convey information to another object with which it has an application-defined relationship, such as a request. This is similar to the *application-initiated* event discussed for Presentation Manager applications in *OS/2 2.1 Volume 4: Writing Applications, Chapter 4 "The Presentation Manager Application Model"*.
- By the user directly manipulating the objects' icons. For example, dropping one icon over another icon initiates a conversation between the two objects. This is similar to the *user-initiated* event discussed for Presentation Manager applications in *OS/2 2.1 Volume 4: Writing Applications, Chapter 4 "The Presentation Manager Application Model"*.

Each of these types of communication is discussed in the following sections.

7.6.1 Application-Initiated Communication

The application-initiated communication is somewhat more complex then the user-initiated communication since the initiator of the communication typically has knowledge of the type of object to which the communication is being passed, and can usually initiate the communication by simply invoking a method in the receiving object, in a similar manner to that discussed in 7.2.1.9, "Invoking Another Object's Methods" on page 113.

However, it is often necessary to determine the identity of the object for which a method must be invoked. The Workplace Shell provides access to objects using the HOBJECT and the OBJECTID and, at a base level, system object model provides pointers to objects and SOM IDs. Each of these is described in the following sections, and some discussion is included on converting between identifiers.

7.6.1.1 HOBJECT

This identifier is the object handle, which is allocated by the Workplace Shell and passed as the return value from the WinCreateObject() function. It is a persistent object handle that remains allocated to an object for the duration of its existence. Object handles persist across system restarts, and may therefore be used by one object to refer to another object at any time.

An object handle can be determined from the object's OBJECTID using the **WinQueryObject()** function,

```
HOBJECT  hObject;                          /* Object handle        */
PSZ      szObjectID = "<OBJECTID>";        /* OBJECTID string      */

hObject = WinQueryObject(szObjectID);      /* Query object handle  */
```

Note that this function may be called from any process; its use is not restricted to objects in the Workplace Shell process.

7.6.1.2 OBJECTID

The OBJECTID is provided by an application or object class as part of the setup string parameter in the WinCreateObject() call, when an object is created. It is persistent in the same way as an object handle, but provides a more meaningful reference for an object, which can be used by other objects.

```
HOBJECT hObject;                                /* Object handle        */

/*********************************************************************/
/* Create a folder on the desktop with an OBJECTID of MYFOLDER       */
/*********************************************************************/

hObject = WinCreateObject("WPFolder",           /* Class Name           */
                          "My Folder",          /* Title                */
                          "OBJECTID=<MYFOLDER>", /* Setup string        */
                          "<WP_DESKTOP>",       /* Location             */
                          CO_REPLACEIFEXISTS);  /* Create option        */

/*********************************************************************/
/* Create a file object with an OBJECTID of MYFILE inside the folder  */
/*********************************************************************/

hObject = WinCreateObject("WPDataFile",         /* Class Name           */
                          "My File",            /* Title                */
                          "OBJECTID=<MYFILE>",  /* Setup string         */
                          "<MYFOLDER>",         /* Location             */
                          CO_REPLACEIFEXISTS);  /* Create option        */

/*********************************************************************/
/* Create a shadow of the file object MYFILE on the desktop           */
/*********************************************************************/

hObject = WinCreateObject("WPShadow",           /* Class Name           */
                          "My File",            /* Title                */
                          "SHADOWID=<MYFILE>",  /* Setup string         */
                          "<WP_DESKTOP>",       /* Location             */
                          CO_REPLACEIFEXISTS);  /* Create option        */
```

Figure 72. Referencing an Object Using OBJECTID

Note that the angle brackets (" < " and " > ") used within the OBJECTID are an important part of the syntax.

Note also that the Workplace Shell provides a number of predefined OBJECTIDs for system-defined objects. The first and third WinCreateObject() calls in Figure 72 use the <WP_DESKTOP> OBJECTID to place the objects on the desktop.

7.6.1.3 SOM Pointer

SOM pointers come in various forms, but can all be typecast to *SOMAny* *. From a Workplace Shell perspective, a SOM pointer is the return value of the _wpclsNew class method; this is the method used for creating objects within the Workplace Shell process. An object's public methods and data can be accessed using the object's SOM pointer.

A SOM pointer for an object may be obtained from an object handle using the _wpclsQueryObject method provided by the *WPObject* class, as follows:

```
SOMAny *Asomptr;                        /* SOM pointers         */
SOMAny *Bsomptr;

Asomptr = _wpclsQueryObject(_WPObject,  /* Query SOM pointer    */
                            hObject);   /* Object handle        */
```

A SOM pointer for a class may be obtained from the SOM ID for that class, using the *_somFindClass* method shown below:

```
Asomptr = _somFindClass(SOMClassMgrObject,
                        AsomId,
                        1,
                        1);
```

A SOM pointer for a class may be obtained from the SOM pointer for any object within that class, using the *_somGetClass* method as follows:

```
Asomptr = _somGetClass(Bsomptr);
```

The SOM pointer is typically used to invoke class methods from an object in another class. The *_SOMDispatchL()* method shown in Figure 45 on page 113 requires a SOM pointer as a parameter.

7.6.1.4 SOM ID

A SOM ID is simply a way of mapping a unique number to a string. This string may represent the name of a method or class. SOM IDs are integers that are managed by the Workplace Shell using the Atom Manager facility of Presentation Manager. A SOM ID is obtained using the **SOM_IdFromString()** function as follows:

```
somId    AsomId;

AsomId = SOM_IdFromString("WPFolder");
```

The SOM ID is typically used to obtain a SOM pointer, which can then be used to invoke a method.

7.6.2 User-Initiated Communication

The user-initiated communication is somewhat more complex then the application-initiated communication, since the two objects may have no defined relationship. A conversation must be initiated between the two, whereby each determines the nature of the other, and whether a drop operation is valid at the present time. If so, each object passes the information required to carry out the requested action.

7.6.2.1 Dragging a Workplace Object over a Workplace Object

When the user begins to drag an object, this source object being dragged is notified by the system, by invoking the object's _wpFormatDragItem method. This method is used to build a DRAGITEM structure, which is passed to any object if this source object is dragged over it or dropped on it. The DRAGITEM structure contains rendering information about the source object, which is used by other receiving objects over which the source object is dragged, in order to determine whether a drop operation is valid at that point.

Default information for the DRAGITEM structure is inserted by the default processing provided by the parent class, but an object class may override the method and include its own class-specific processing. The DRAGITEM structure is nested within a DRAGINFO structure, which is passed to any receiver object over which one or more source objects are dragged. In a situation where more than one object is being dragged simultaneously, a separate DRAGITEM structure is produced for each source object, and the entire set of structures is combined using a single DRAGINFO structure.

When an object is dragged over an object the system invokes the _wpDragOver method in the receiving object. This method receives a DRAGINFO structure, which contains a variety of information including pointers to one or more DRAGITEM structures. Note it is the responsibility of the receiver object's _wpDragOver method to return whether it can accept the drop, and what operation to perform.

The receiver object has to check the operation code it receives from the source object. If it is the default (*DO_DEFAULT*), it needs to return the operation to be performed, for example *DO_MOVE*, or *DO_COPY*. If it is not the default, for example a *DO_MOVE*, then the receiver object must determine if it can accept that operation and return accordingly.

Warning

If you have written Workplace Shell drag and drop code under OS/2 2.0, the way the ancestor classes respond to the _wpDragOver method has changed for OS/2 2.1. Specifically the parent_wpDragOver method will return *DOR_NEVERDROP* if the ancestor classes cannot accept the source objects.

Consequently code must be written so that the parent_wpDragOver method is called first, and if the resultant Drop Indicator is ***NOT*** *DOR_NEVERDROP*, then the method may continue its processing, as shown in Figure 73 on page 154.

Figure 73 on page 154 shows another _wpDrop method override example for the object called pwFinanceFile which is derived from the wpDataFile Workplace Shell object. In this example we first invoke our objects parent _wpDragOver method (that is the _wpDragOver method for wpDataFile) to see that it doesn't violate any of its rules. If the parent successfully tested one or more source objects, our method determines how many source objects are being dragged over the target object and then searches through each of the source objects() and checks that they are all Workplace Objects. If any one of these is not a Workplace Object then our object will reject all of them by returning *DOR_NEVERDROP* as the indicator and *DO_UNKNOWN* as the operation. Otherwise our method makes sure it is being passed a *DO_COPY*, *DO_MOVE* or a *DO_DEFAULT* operation. If it is not one of these, it will reject all of the source objects. If it is one of these, *DO_DROP* and *DO_COPY* will be returned as the result to the Workplace Shell.

If the receiver object wanted to allow only its tests and not any of the parents, then the parent_wpDragOver method invocation can be removed, along with the setting of the *dropIndication* and *dropOperation* from the returned *mr* program variable, as well as the *if* statement that immediately follows.

If the _wpDragOver in Figure 73 on page 154 returns successful, and the user drops the object, then the _wpDrop method is invoked for the receiver object. The same checking that was performed in the _wpDragOver method needs to be performed by the _wpDrop method because the receiving object may only receive a _wpDrop method invocation, and not a _wpDragOver.

```
SOM_Scope MRESULT   SOMLINK pwFinanceFile_wpDragOver(PWFinanceFile *somSelf,
                HWND hwndCnr,
                PDRAGINFO pdrgInfo)
{
    MRESULT  mr;
    USHORT   dropOperation,
             dropIndicator;
    ULONG    ulItemCount        =0;
    ULONG    ulItem             =0;

    PWFinanceFileData *somThis = PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile","pwFinanceFile_wpDragOver");

    /* firstly will all the source object(s) pass my parents tests? */
    mr = parent_wpDragOver(somSelf,hwndCnr,pdrgInfo);
    dropIndicator = SHORT1FROMMR(mr);
    dropOperation = SHORT2FROMMR(mr);

    if (dropIndicator != DOR_NEVERDROP)
    {
      /* passed the parent's tests, so unless it fails this object's */
      /* tests we will allow the DROP                                */
      dropIndicator = DOR_DROP;
      dropOperation = DO_COPY;

      /* how many items are being dragged ? */
      ulItemCount = DrgQueryDragitemCount(pdrgInfo);

      /* search through the objects and abort if we find any that */
      /* are not WorkPlace objects                                */

      for (ulItem=0; ulItem<ulItemCount; ulItem++) {
        PDRAGITEM pDragItem;              /* temporary variable */
        WPObject *ObjectBeingDragged=NULL; /* temporary variable */

        /* get one of the one or more drag items that we are receiving */
        pDragItem = DrgQueryDragitemPtr(pdrgInfo, ulItem);

        /* test to see if it is a WorkPlace object, if it is use     */
        /* the OBJECT_FROM_PREC macro to get the object; otherwise   */
        /* ObjectBeingDragged will remain as a NULL as it was        */
        /* initialised when declared as NULL                         */
```

Figure 73 (Part 1 of 2). Dragging a Workplace Object

```
        if ( DrgVerifyRMF(pDragItem, "DRM_OBJECT", NULL))
           ObjectBeingDragged = OBJECT_FROM_PREC(pDragItem);
        if (!ObjectBeingDragged) {
            /* Object is NOT a WorkPlace object, so I reject all objects */
            return (MRFROM2SHORT(DOR_NEVERDROP,DO_UNKNOWN));
          } else {
            if ( (pdrgInfo->usOperation != DO_COPY   ) &&
                 (pdrgInfo->usOperation != DO_MOVE   ) &&
                 (pdrgInfo->usOperation != DO_DEFAULT) ) {
            /* this object only allows a move or copy */
            return (MRFROM2SHORT(DOR_NODROP,DO_UNKNOWN));
            } /* endif */
          } /* endif */

        } /* endfor */
    } /* endif */
    /* all the test have been passed, so tell this to the WorkPlace Shell */
    return (MRFROM2SHORT(dropIndicator, dropOperation));
 }
```

Figure 73 (Part 2 of 2). Dragging a Workplace Object. In this figure we first see if the source's object passes this object's parent's wpDragOver tests, and then apply our own.

Note that *DOR_NODROP* is returned when rejecting the operation, and *DOR_NEVERDROP* is returned when rejecting the drop request. This is important when a user drags one object onto another (DO_MOVE) and the receiver object returns a *DOR_NODROP*, meaning it cannot accept the operation. If the user still has the object over the receiver object and now holds down the <ctrl> key, then the operation is now a DO_COPY. If the receiver object had previously returned a *DOR_NODROP* then the Workplace Shell will now reinvoke the objects _wpDragOver method passing it the DO_COPY. This would not occur if a *DOR_NEVERDROP* had been received.

How the receiver object responds with its implementation of the _wpDragOver, really depends on:

- The type of object being implemented
- What is being dragged over it
- What the person who designs the object really wants to achieve

For example, in Figure 73 on page 154, simple decisions are made based on the parent's _wpDragOver method. Only Workplace Objects which want to perform a *DO_COPY*, *DO_MOVE* or a *DO_DEFAULT* operation are acceptable source objects.

The designer may want to restrict what the object will accept still further. For example, only allowing objects that are the same class as the pwFinanceFile object, or are descended from it, as there are going to be specific data types that have meaning to this object and no others can be accepted. Perhaps the designer wishes to have financial account type entries in the pwFinanceFile object and it would therefore not be meaningful to allow the dragging and dropping of a picture or any other objects that could not be understood by this specialized object.

Figure 74 on page 156 expands on Figure 73 on page 154 to now exclude any objects that are not part of the pwFinanceFile class.

```
SOM_Scope MRESULT   SOMLINK pwFinanceFile_wpDragOver(PWFinanceFile *somSelf,
                HWND hwndCnr,
                PDRAGINFO pdrgInfo)
{
    MRESULT  mr;
    USHORT   dropOperation,
             dropIndicator;
    ULONG    ulItemCount        =0;
    ULONG    ulItem             =0;
    CLASS    PWFinanceFileClass;

    PWFinanceFileData *somThis = PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile","pwFinanceFile_wpDragOver");

    /* create a dummy pwFinanceFile class for comparison with the source */
    /* object(s) class                                                   */
    PWFinanceFileClass = _somClassFromId( SOMClassMgrObject,
                         SOM_IdFromString("PWFinanceFile") );

    /* firstly will all the source object(s) pass my parents tests? */
    mr = parent_wpDragOver(somSelf,hwndCnr,pdrgInfo);
    dropIndicator = SHORT1FROMMR(mr);
    dropOperation = SHORT2FROMMR(mr);

    if (dropIndicator != DOR_NEVERDROP)
    {
      /* passed the parent's tests, so unless it fails this object's */
      /* tests we will allow the DROP                                */
      dropIndicator = DOR_DROP;
      dropOperation = DO_COPY;

      /* how many items are being dragged ? */
      ulItemCount = DrgQueryDragitemCount(pdrgInfo);

      /* search through the objects and abort if we find any that */
      /* are not WorkPlace objects                                 */

      for (ulItem=0; ulItem<ulItemCount; ulItem++) {
        PDRAGITEM pDragItem;              /* temporary variable */
        SOMAny   *ObjectBeingDragged=NULL; /* temporary variable */

        /* get one of the one or more drag items that we are receiving */
        pDragItem = DrgQueryDragitemPtr(pdrgInfo, ulItem);

        /* test to see if it is a WorkPlace object, if it is use     */
        /* the OBJECT_FROM_PREC macro to get the object; otherwise   */
        /* ObjectBeingDragged will remain as a NULL as it was        */
        /* initialised when declared as NULL                         */

        if ( DrgVerifyRMF(pDragItem, "DRM_OBJECT", NULL))
           ObjectBeingDragged = OBJECT_FROM_PREC(pDragItem);
```

Figure 74 (Part 1 of 2). Only Accepting pwFinanceFile Objects from Drag Operations

```
        if (!ObjectBeingDragged) {
            /* Object is NOT a WorkPlace object, so I reject all objects */
            return (MRFROM2SHORT(DOR_NEVERDROP,DO_UNKNOWN));
          } else {
            if ( (pdrgInfo->usOperation != DO_COPY   ) &&
                 (pdrgInfo->usOperation != DO_MOVE   ) &&
                 (pdrgInfo->usOperation != DO_DEFAULT) ) {
            /* this object only allows a move or copy */
            return (MRFROM2SHORT(DOR_NODROP,DO_UNKNOWN));
            } else {
              /* the object is all ok, but is it a pwFinanceFile object,  */
              /* or descended from one?                                   */
              if (!_somIsA(ObjectBeingDragged,PWFinanceFileClass)) {
                /* reject all the objects because this one is not derived */
                /* from PWFinanceFileClass                                 */
                return (MRFROM2SHORT(DOR_NEVERDROP,DO_UNKNOWN));
              } /* endif */

            } /* endif */

          } /* endif */

        } /* endfor */

    } /* endif */
    /* all the test have been passed, so tell this to the WorkPlace Shell */
    return (MRFROM2SHORT(dropIndicator, dropOperation));
}
```

Figure 74 (Part 2 of 2). Only Accepting pwFinanceFile Objects from Drag Operations

7.6.3 Dragging a Non-Workplace Object onto a Workplace Object

Dragging a non-Workplace Object onto a Workplace Object is handled in a similar way to the dragging of a Workplace Object over a Workplace Object. How the receiver object responds in its implementation of its _wpDragOver, depends on the type of object being implemented, what is being dragged over it and what the person who designed the object wants it to do. The difficulty is to determine what the object should do for every possible type of non-Workplace Shell object (and Workplace Shell objects such as OS/2 files).

If the receiver object is capable of having a file object dropped on it, and a file was dragged over it (the source is not a Workplace Object file object), the receiver object could convert the source to a file object and then process it.

If the source of the drag operation cannot be converted to a file object, or is not a file in the first place, then drag and drop participation is still possible if the source object has a drag mechanism and operation that the target object can support. 7.6.5, "Dropping an Object" on page 159 shows an example of a _wpDrop method where an OS/2 file is accepted and converted to a Workplace Object.

7.6.4 Dragging a Workplace Object onto a Non-Workplace Object

In the same way a non-Workplace Object can participate in a drag and drop conversation with a Workplace Object, a Workplace Object can can also participate in a drag and drop conversation with a non-Workplace Object, provided the Workplace Object provide support for the drag mechanisms and operations that the receiver object supports.

A Workplace Object can support multiple mechanisms and operations as shown in Figure 75.

```
   MRESULT       mr;
   PDRAGTRANSFER pDragTransfer;
   BOOL          bSentOK;
      .
      .
      .
   /* allocate a drag transfer structure */
   pDragTransfer = DrgAllocDragtransfer(1);

   if (pDragTransfer) // was the allocate successful?
   {
      /* populate the drag structure */
      .
      .
      .
      pDragTransfer->hstrSelectedRMF  =
          DrgAddStrHandle("(DRM_OS2FILE,DRM_PRINT)x(DRF_TEXT),
                           <DRM_OBJECT,DRF_OBJECT>");
      .
      .
      .
      bSentOK = (BOOL)DrgSendTransferMsg(pDragInfo->hwndSource,
                                         DM_RENDERPREPARE,
                                         (MPARAM)pDragTransfer,
                                         (MPARAM)NULL);
      if (bSentOK)
      {
        .
        .
        .
      } else {
        mr = (MRESULT)RC_DROP_ERROR;
      }
        .
        .
        .
}
```

Figure 75. Multiple Rendering Methods

7.6.5 Dropping an Object

When a drop operation occurs, the receiver object is notified by the system which invokes the _wpDrop method for that object. This method accepts the DRAGINFO structure which may then be examined by the receiver object to determine the correct action to be taken. The rendering information contained in the DRAGITEM structure may be sufficient to allow the action to be completed, or the receiver object may initiate a conversation with the source object in order to gain sufficient information to complete the action.

If the source object is not a Workplace Object but it is an OS/2 file, then the receiver object must decide whether it can handle a file. If receiver handling is set then the receiver object can create a Workplace Object created to represent the OS/2 file. This Workplace Object can then issue methods. However, if you do not wish to create a Workplace Object to represent the file then the Presentation Manager Drag and Drop messages must be handled.

If neither a Workplace Object, nor an OS/2 File are being dropped, then the receiver object must decide what it wants to do.

The rendering information provided in the DRAGITEM structure, and its use by a Presentation Manager or Workplace Shell object, is described in detail in *OS/2 2.1 Volume 4: Writing Applications, Chapter 8 "Direct Manipulation."*

```
/*
 *
 *   METHOD:   wpDrop                                        PUBLIC
 *
 *   PURPOSE:  To receive a dropped object.
 *
 *   INVOKED:  By Workplace Shell, when another object has been dropped on
 *             this object.
 *
 */

SOM_Scope MRESULT   SOMLINK pwFinanceFile_wpDrop(PWFinanceFile *somSelf,
          HWND hwndCnr,
          PDRAGINFO pdrgInfo,
          PDRAGITEM pdrgItem)
{

    CHAR          szNameffCCHMAXPATH";
    CHAR          szPathffCCHMAXPATH";
    ULONG         cbPath = CCHMAXPATH;
    ULONG         ulItemCount         =0;
    ULONG         ulItem              =0;
    CLASS         PWFinanceFileClass;
    SOMAny       *ObjectBeingDragged;
    MRESULT       mr;
    USHORT        dropOperation,
                  dropIndicator;
    BOOL          flPrepared = TRUE; // Assume we do not need to do a prepare
    BOOL          flRendering = FALSE;
    PDRAGTRANSFER pDragTransfer;

    PWFinanceFileData *somThis = PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile","pwFinanceFile_wpDrop");

    if ((strcmp(_szCurrentPassword,       /* If FinanceFile is NOT locked */
                _szPassword)) == 0)
    {

  /* make sure we are not dragging ourselves, and dropping onto ourselves */
     if (pdrgInfo->hwndSource != hwndCnr)
     {

     /* for each of the items being dropped, check to see that they are all */
     /* derived from PWFinanceFileClass                                     */

     PWFinanceFileClass = _somClassFromId( SOMClassMgrObject,
                          SOM_IdFromString("PWFinanceFile") );
```

Figure 76 (Part 1 of 5). Converting a Source Drag OS/2 File to a Workplace Object

```
/* passed the parent's tests, so unless it fails this object's */
/* tests we will allow the DROP                                */
dropIndicator = DOR_DROP;
dropOperation = DO_COPY;

/* how many items are being dragged ? */
ulItemCount = DrgQueryDragitemCount(pdrgInfo);

/* search through the objects and abort if we find any that aren't derived */
/* from PWFinanceFileClass                                                  */
somPrintf("Number of Items being dropped = %i.\n",ulItemCount);
for (ulItem=0; ulItem<ulItemCount; ulItem++) {
  PDRAGITEM pDragItem; /* temporary variable*/

  /* get one of the one or more drag items that we are receiving */
  pDragItem = DrgQueryDragitemPtr(pdrgInfo, ulItem);

  ObjectBeingDragged = queryObjectFromDragItem(pDragItem);

  if (ObjectBeingDragged) {
    if (!_somIsA(ObjectBeingDragged,PWFinanceFileClass)) {
      somPrintf("Object %i, is rejected for drop because it ",ulItem);
      somPrintf("is not derived from PWFinanceFileClass\n");
    } else {
      somPrintf("Object %i, is acceptable for dropping, by wpDROP\n",ulItem);
    } /* endif */
  } else {
    somPrintf("Object %i, is not a WPS object, can we render it\n",ulItem);

/* start of code to render item */

if( DrgVerifyRMF (pDragItem, "DRM_OS2FILE", NULL) )
{
   somPrintf("An OS2FILE rendering method!\n");
   /* Protocol allows the source object to propose a target name...
    *
    * If it does, then try to use it, if it does not, then
    * try to use the source name, if present.  Finally, just
    * make up our own name...
    */
   if (pDragItem->hstrTargetName &&
       DrgQueryStrNameLen(pDragItem->hstrTargetName) )
   {
      DrgQueryStrName(pDragItem->hstrTargetName,
                      sizeof(szName),szName);
      somPrintf("Source proposes the target filename\n");
   }
   else
   {
```

Figure 76 (Part 2 of 5). Converting a Source Drag OS/2 File to a Workplace Object

```
   if (pDragItem->hstrSourceName &&
      DrgQueryStrNameLen(pDragItem->hstrSourceName))
   {
      DrgQueryStrName(pDragItem->hstrSourceName,
                      sizeof(szName),szName);
   somPrintf("Source proposes the source filename\n");

   }
   else
   {
      szNameffl0" = '\0';
      somPrintf("no source, nor target name\n");
   }
}

/* Allocate and initialize a drag transfer structure
 */
somPrintf("allocating pDragtransfer structure\n");
pDragTransfer = DrgAllocDragtransfer(1);

if (pDragTransfer)
{
   somPrintf("pDragtransfer structure allocated ok\n");
   /* create a WPDataFile object
    */
   ObjectBeingDragged = _wpclsNew( _WPDataFile,
                      szName,
                      NULL,
                      _wpclsQueryFolder(_WPDataFile,"<WP_NOWHERE>",TRUE),
                      TRUE );

   if (ObjectBeingDragged)
   {
      somPrintf("ObjectBeingDragged has been successfully allocated\n");
      _wpQueryRealName(ObjectBeingDragged,szPath,&cbPath,TRUE);
      somPrintf("The ObjectBeingDragged filename is %s\n",szPath);

      /* fill in the struct now
       */
      pDragTransfer->cb                = sizeof(DRAGTRANSFER);
      pDragTransfer->hwndClient        = hwndCnr;
      pDragTransfer->pditem            = pDragItem;
      pDragTransfer->hstrSelectedRMF   =
                     DrgAddStrHandle("<DRM_OS2FILE,DRF_UNKNOWN>");
      pDragTransfer->hstrRenderToName = DrgAddStrHandle( szPath );
      pDragTransfer->ulTargetInfo      = 0L;
      pDragTransfer->usOperation       = pdrgInfo->usOperation;
      pDragTransfer->fsReply           = 0;
```

Figure 76 (Part 3 of 5). Converting a Source Drag OS/2 File to a Workplace Object

```
/* Now, if the source wants prepared, do it...
 */
if (pDragItem->fsControl & DC_PREPARE)
{
   somPrintf("Source wants prepared\n");
   flPrepared = (BOOL)DrgSendTransferMsg(pdrgInfo->hwndSource,
                                         DM_RENDERPREPARE,
                                         (MPARAM)pDragTransfer,
                                         (MPARAM)NULL);
} else {
   somPrintf("Source does not want prepared\n");
}
/* See if either we did not need to send a RENDERPREPARE, or
 * we have successfully done so...
 */

if (flPrepared)
{
   somPrintf("not prepared\n");
   /* Tell the source object where to put the file.
    */
   flRendering = (BOOL)DrgSendTransferMsg(pDragItem->hwndItem,
                                          DM_RENDER,
                                          (MPARAM)pDragTransfer,
                                          (MPARAM)NULL);
   if (!flRendering)
   {
      /* The partner object did not render, so delete
       * the object we just created.
       * or we could add code here to directly open the source as
       * a file and work with it, or what ever we like.
       */

      _wpFree(ObjectBeingDragged);
      somPrintf("not rendering, we are deleting the object we just created\n");
   } else {
      somPrintf("rendering\n");
   }
}
else
{
```

Figure 76 (Part 4 of 5). Converting a Source Drag OS/2 File to a Workplace Object

```
                somPrintf("Our partner wanted us to send him a prepare, and");
                somPrintf("now has changed his mind about things..., ABORT\n");

                /* Our partner wanted us to send him a prepare, and
                 * now has changed his mind about things...
                 * We cannot even send him an end conversation, as
                 * we do not know that the hwnd is any good.
                 *
                 * For now, we will treat this as an error.
                 */
                mr = (MRESULT)RC_DROP_ERROR;
              }
            } else {
              somPrintf("ObjectBeingDragged has NOT been successfully allocated\n");
            }
            if (flRendering)
            {
              mr = RC_DROP_RENDERING;
            }
            else
            {
              DrgDeleteStrHandle( pDragTransfer->hstrRenderToName );
              DrgFreeDragtransfer( pDragTransfer );
            }
          }
        } else {
          somPrintf("Not an OS2FILE rendering method\n");
        }

          } /* endif */

        } /* for */
      } else {
        somPrintf("we are trying to drop onto ourselves, not allowed\n");
      } /* endif */
    } else {
      somPrintf("LOCKED, drop is disallowed\n");
    } /* endif */

    return((MRESULT) NULL);

  }
```

Figure 76 (Part 5 of 5). Converting a Source Drag OS/2 File to a Workplace Object. A partial sample _wpDrop accepting a non-Workplace Object, specifically an OS/2 file.

7.7 Building a Workplace Shell Application

As already mentioned, an application that exploits the Workplace Shell consists of a number of objects on the desktop or in folders, which interact with one another to carry out operations as requested by the user. The implementation of the Workplace Shell under OS/2 V2.0 causes all Workplace Shell objects to run in a single process, under the control of the Workplace Shell itself. It is therefore possible for an error in a Workplace Shell to terminate the Workplace Shell process, and all objects currently open under the control of that process. While the Workplace Shell automatically restarts itself and its open objects, it is recommended for reasons of performance that applications carrying out lengthy processing such as database or remote system access should be implemented using multiple processes. Other processes in the system are not affected if the Workplace Shell process terminates, and become available to the user as soon as the shell restarts itself, without the need to reload application code, reinitialize communications links, etc.

For example, a database query application that searches a database for customer records and displays these in a Workplace Shell folder may be composed of two processes, each with multiple threads, as shown in Figure 77 on page 165.

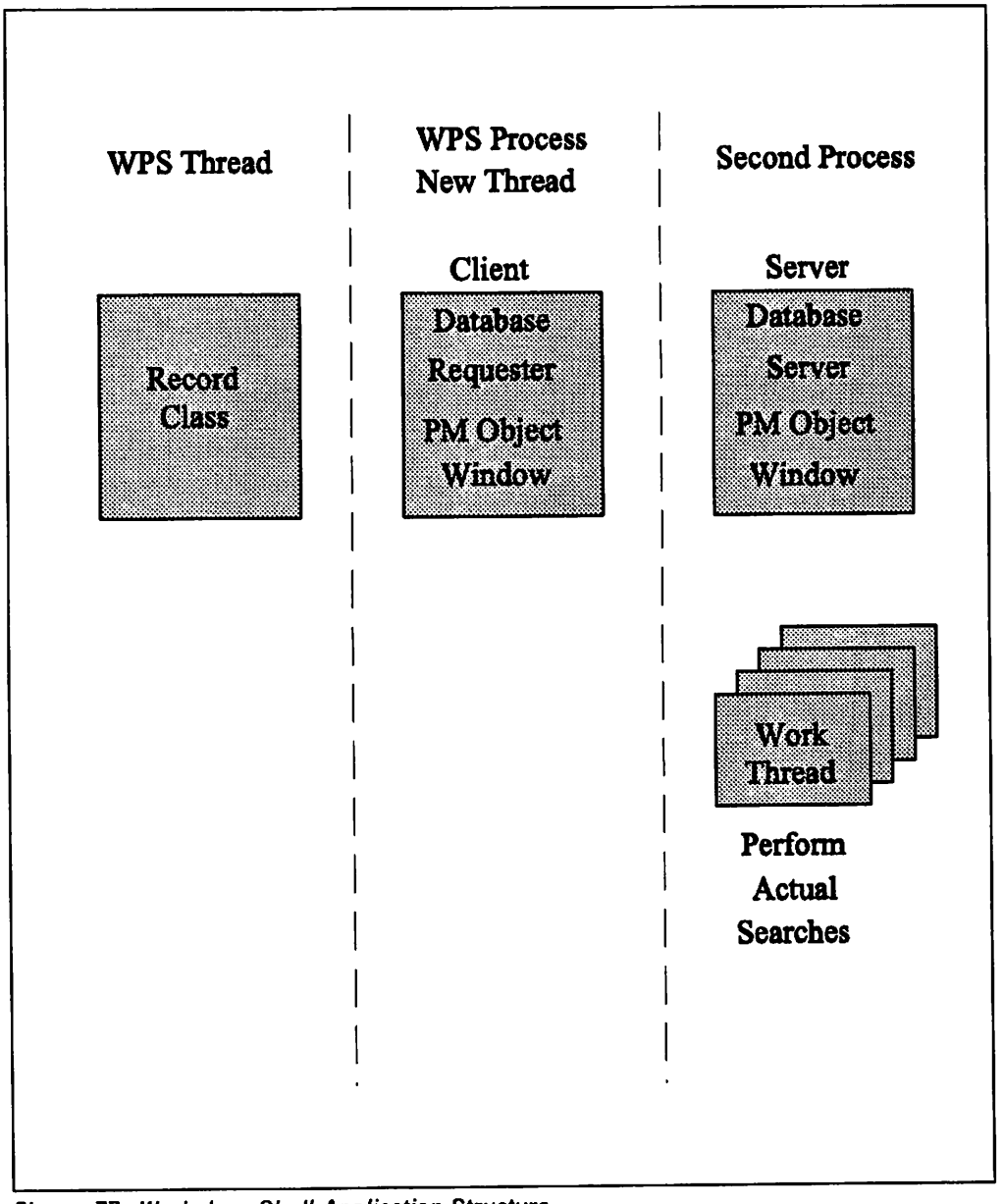

Figure 77. Workplace Shell Application Structure

The requester portion of the application, which allows the user to enter a query, and which displays the results on the screen, is implemented as a Workplace Shell object, running under the control of the Workplace Shell process. The primary thread in this process carries out the interaction with the end user, while a secondary thread is created to handle communication between processes.

The second process acts as a database server, and is created by the first process when the application is started. The server process has a primary thread that accepts requests from the requester in the Workplace Shell process, and a number of secondary threads that actually perform the database access.

If an errant object or application were to cause the Workplace Shell to terminate, the requester threads would be terminated. However, the server process *would not* be terminated, and communication with the requester could be re-established simply by having the requester initiate one of the standard interprocess communication mechanisms described in *OS/2 2.1 Volume 4: Writing Applications, Chapter 10, "Multitasking Considerations"*.

7.8 Debugging

SOM provides several facilities to aid in debugging SOM and WPS applications. These facilities are designed around a replaceable procedure called *SOMOutCharRoutine*, which normally writes output to the *stdout* logical device.

It is not practical to capture *stdout* and the information that SOM and WPS is providing when debugging WPS objects, because objects are implemented as DLLs. Instead the object must replace the *SOMOutCharRoutine* to send the output to a place where you can easily deal with it.

Additionally the OS/2 Programmer's Toolkit provides an interactive debugging tool, the Kernel Debugger. For further information on using the Toolkit Kernel Debugger to debug DLLs, please refer to the *OS/2 Programmer's Toolkit.*

7.8.1 Replacing SOM's SOMOutCharRoutine

Figure 78 on page 167 shows a simple replacement procedure that sends the output to the *COM1:* serial port. To actually replace the *SOMCharOutRoutine* it is necessary to add a line of code to the initialization portion of your object. This is shown in Figure 79 on page 167. Note that it is best to add a command to your *STARTUP.CMD* file to set up the *COM1:* serial port. A sample portion of a *STARTUP.CMD* is shown in Figure 80 on page 168.

You can now use another computer to view the information that is generated by the SOM runtime as you manipulate your object.

The additional computer should be connected from its serial port, normally COM1:, to the first computer by means of a NULL modem cable. This is a specialized serial cable where the transmit and receive conductors are crossed over to allow the transmission from one serial port to be received by the other.

It is then a matter of registering your object and running an ASCII terminal emulator program on the additional computer to view the information.

```
    .
    .
    .
#
# Passthru a debug message box to the .ih file
# (for inclusion in the .c file)
#
passthru: C.ih, after;

#include <wppgm.h>
#include <wppgmf.h>
#include <stdio.h>

// force SOM to output all debug information to the Communications Port 1

int myReplacementForSOMOutCharRoutine (char c)
{
static FILE *fdebug = NULL;

if (!fdebug) {
  fdebug = fopen("COM1","w");

  if (!fdebug) return 0;     /* failed to open COM1: */
  }
fputc(c,fdebug);
fflush(fdebug);

return 1;
}
endpassthru;
    .
    .
    .
```

Figure 78. Sample .CSC File Definition for Overriding the SOMOutCharRoutine

```
    .
    .
    .
    /* Set up the debug and tracing ... */

    /* Produce a message each time a method is entered */
    SOM_TraceLevel=2;

    /* Replace the default routine with this object's new one */
    SOMOutCharRoutine = myReplacementForSOMOutCharRoutine;
    .
    .
    .
```

Figure 79. Sample .C File Definition for Overriding the SOMOutCharRoutine

```
/* My STARTUP.CMD */
MODE COM1 9600,n,8,1
/* and whatever else I like to have in here */
    .
    .
    .
exit 0
```

Figure 80. Sample STARTUP.CMD File Definition

7.8.2 A Sample ASCII Terminal Emulator for Debugging Use

The PM terminal program that can be found in the Productivity folder can be used to receive and display the information from the SOM runtime about the computer to be debugged. The following steps detail how to create and use a custom emulator session.

7.8.2.1 Creating a Custom Emulator Session

The following steps detail how to create a PM Terminal session suitable for remote debugging use.

1. Open **OS/2 System** and select the **Productivity** Folder
2. Select **PM Terminal**
3. From the **Session** pulldown, select **Add**
4. Enter a comment, for example "Remote SOM Debug Session"
5. You should have the default settings as shown in Table 4, on the **Add Session** panel
6. Select the **ADD** pushbutton to add the new session

Table 4. Parameters and Settings for the Remote Terminal

Parameter	Setting
Terminal emulation profile	ANSI 3.64
Connection path profile	ACDI - Hardwire
System environment profile	Default Environment
File transfer profile	Character

7.8.2.2 Using the Emulator Session

After you have created an instance of your object, start the session you created above. If the connection is broken at any time, for example you closed and reopened the object, then from the **File** pulldown select **Connect**.

7.8.3 SOM Provided Macros for Debugging

System Object Model provides a number of macros for the purposes of debugging objects. These are:

SOM_TestC Evalutes a Boolean expression. If it is true, then execution continues; otherwise **SOM_Error** is invoked.

SOM_WarnMsg Writes out a warning message depending on the setting of the SOM_WarnLevel variable.

SOM_Assert Evaluates a Boolean assertion. If this fails then SOM_Error is invoked with a user supplied error code.

SOM_Expect Evaluates a Boolean assertion. If this fails then SOM_Warn is invoked.

somPrintf SOM's implementation of the "C" printf function.

For more detailed information please refer to *System Object Model Guide and Reference.*

7.9 Sample Code and Application

The sample Workplace Objects used as examples in this chapter are included on a diskette supplied with this document, as well as the main program listings appearing in Appendix E, "Source Code for the PWFolder and PWFinanceFile objects" on page 347. The two main code examples are pwFolder and pwFinanceFile.

7.9.1 pwFolder

The pwFolder is a Workplace Object that has been sub-classed from the Workplace Folder class and adds a lock feature. This lock feature prevents the user from accessing the folder when it is in the locked state.

The pwFolder demonstrates adding a method to a context menu and writing your own methods, as well as drag and drop processing.

7.9.2 pwFinanceFile

The pwFinanceFile is a Workplace Object that has been subclassed from the wpDataFile class and also adds a lock feature, in a very similar manner to the pwFolder.

The pwFinanceFile demonstrates the following:

- Adding a method to a context menu
- Filtering menu items in a context menu
- Writing your own methods
- Drag and drop progressing, including:
  - Only accepting source objects which are descended from the same pwFinanceFile class
  - Accepting an OS/2 file as a source, and converting it to a Workplace Object
  - Multiple rendering methods
- Adding an object view
- Determining the file name of a wpDataFile object

7.10 Summary

The OS/2 Version 2.0 Workplace Shell provides an object-oriented user interface to the operating system, and provides an object-oriented application layer on top of Presentation Manager, through its implementation of the system object model. An application that exploits the facilities of the Workplace Shell consists of a number of objects, which are manipulated on the Workplace Shell desktop by the user, in order to carry out the required tasks.

Workplace Shell objects conform to "standard" object-oriented principles in that they are grouped into object classes, have instance data, and contain methods which perform the required tasks and operate upon the instance data. Workplace Shell object classes may inherit data and methods from their parent class, in accordance with the object-oriented concept of inheritance. A class may add additional data items or new methods to perform actions not handled by its parent class, or may override existing methods to handle actions in a different manner.

An object class is defined using a class definition file, which defines the parent hierarchy for the object, its data items and its methods. The class definition file is used as input to the SOM Precompiler, which uses the file to produce a number of source code files and header files. The source code is edited by the programmer to add the application logic for each method. It is then compiled using a normal C compiler, and link edited to produce a dynamic link library that implements the object class. An object class may make use of operating system and Presentation Manager resources during its execution.

Workplace Shell objects behave in a similar manner to windows under Presentation Manager; object classes are registered to the Workplace Shell, and individual instances of an object class are created, opened, closed and destroyed by the user or by other objects in the system. The major difference between a window under Presentation Manager and an object under the Workplace Shell is that Workplace Shell objects are persistent; that is, they exist for longer than a single application execution. Once created, a Workplace Shell object remains in existence until it is explicitly destroyed.

Chapter 8. Direct Manipulation

Direct manipulation of icons on the Presentation Manager desktop in order to carry out processing tasks has been possible since the first release of Presentation Manager in OS/2 Version 1.1, but only in Version 1.3 was a standard method introduced for implementing such function. Previously, each programmer needed to devise a set of protocols, and write code in every application to handle all the mouse messages and interaction between windows that may have been needed.

OS/2 Version 1.3 introduced some standards for coding direct manipulation operations, in the form of new message classes (the DM_xxxx messages), and standard protocols known as **rendering mechanisms**, which are used to communicate required information for commonly used direct manipulation operations. This support is continued in OS/2 V2.0, and is of increased importance since the Workplace Shell itself makes extensive use of direct manipulation. Applications that use direct manipulation are therefore more likely to be written under Version 2.0, either to interact with one another or to make use of Workplace Shell facilities such as printer objects or the shredder.

This chapter discusses the use of direct manipulation in a program, covering the major messages and data structures involved, and the use of the standard rendering mechanisms. Examples of the use of these messages and data structures are given, along with guidance on implementing a private rendering mechanism to meet the needs of a particular application.

8.1.1 Direct Manipulation Basics

It might appear that dragging an icon from one window and dropping it onto another is straightforward, but on closer consideration it proves to be somewhat more complex. Consider the simple action of dragging an icon representing a customer from one container window to another, the intention being to move the customer from one branch of the business (represented by the first container) to another (the second container window). The following steps are required:

1. The program owning the first container (hereafter called the **source** program) must determine which customer the user wishes to move (hereafter known as the **dragitem**).
2. The source program must decide on an icon or bitmap to represent the customer as the user drags the customer around the screen.
3. The source program must package a number of items of information to travel with the icon, so that potential target windows may decide whether or not they will allow the item to be dropped on them.
4. As the icon passes outside the container to other areas, its appearance must be constantly updated to indicate to the user whether a drop is allowed.
5. When the icon reaches a potential target window, the program owning the target window (hereafter known as the **target**) must access the information about the dragged item to decide whether it will allow the item to be dropped. At this point, the rendering mechanism used to convey this information becomes significant, since both the source and target must be able to understand the mechanism.

6. Once a drop has occurred, the target window must decide the form in which it wishes to receive the dropped object (if more than one form is supported by both source and target), and inform the source program accordingly.
7. The source program must make the data representing the customer available to the target program. Since the source and target programs may not necessarily run in the same process, this may not be trivial. Again, the rendering mechanism to be used becomes significant.
8. The source must inform the target that the data is ready.
9. The target must access and retrieve the data.
10. The source must delete its own copy (since the operation is a "move" operation).
11. The target must display the new customer object in its own container window, in the location at which it was dropped.

While this appears extremely complex, much of the necessary work is done by Presentation Manager for a Presentation Manager application; for a Workplace Shell object, even more of the necessary function is automated by the Workplace Shell. The remainder of this chapter will describe the steps necessary for a Presentation Manager application and/or a Workplace Shell object to exploit drag/drop functionality.

8.1.2 Significant Events

There are a number of significant events that occur during a direct manipulation operation, and which must be communicated to the source and/or target of the operation. These are as follows:

- Initiation of the drag operation, which must be communicated to the source so that it can initialize data structures with information relating to the dragitem.
- Dragging the object over a potential target, which must be communicated to the target so that the target can determine whether a drop operation is valid.
- Dropping the object over a target, which must be communicated to the target so that it may decide the form in which it wishes the dragitem data to be passed, and allocate any necessary resources to receive the data.
- Transferring the information between the source and the target to complete the overall direct manipulation sequence.

Presentation Manager carries out much of the required notification of drag/drop events using messages, which are passed to the source or target windows as necessary during the drag/drop operation. The required messages are described in the *OS/2 2.0 Programming Guide Volume II* and their use is discussed in 8.3, "Using Direct Manipulation" on page 177.

In certain cases, the behavior of a Workplace Shell object participating in a direct manipulation operation varies somewhat from that of a Presentation Manager window. This is due to the fact that the Workplace Shell implements much of the required message handling itself, and directly invokes the appropriate methods in the object. Where the behavior of a Workplace Shell object differs from that of a Presentation Manager window, this is noted in the text.

8.1.3 Rendering Mechanisms

Rendering mechanisms are the means by which the source and target of a drag/drop operation determine the data type of the dragitem and the format of the information to be exchanged.

While the precise sequence of events that takes place after a drop has occurred is dependent upon the application, a number of standard rendering mechanisms have been defined to enable diverse applications to engage in direct manipulation with one another. These standard rendering mechanisms are used by various components of OS/2, as well as being available for use by applications.

Three standard rendering mechanisms are provided by Presentation Manager and are documented in the *OS/2 2.0 Programming Guide Volume II*:

DRM_PRINT This rendering mechanism is designed for applications that wish to provide printing facilities via direct manipulation, by allowing the user to drag items from the application and drop them on one of the Workplace Shell printer objects.

It is a very simple mechanism. When an object is dropped on a printer object, the printer object sends a DM_PRINTOBJECT message, one parameter of which gives the name of the print queue represented by that printer object. It is then the responsibility of the source window to print the relevant data to the specified queue.

Note that in OS/2 Version 1.3, the DM_PRINT message was used for this purpose, rather than the DM_PRINTOBJECT message.

DRM_OS2FILE This rendering mechanism is intended for applications that wish to allow the dragging and dropping of file objects between windows or folders on the desktop. Such applications include the File Manager in OS/2 Version 1.3, or the *Drives* objects in OS/2 V2.0.

With this mechanism, all information about the source file may be contained in the fields of the DRAGITEM structure, so it is not necessary for a protracted conversation to take place between source and target windows. In the simplest case, the target can complete the operation using only this information with no further involvement from the source window, though this rendering mechanism does allow for more interaction between the two windows should this be useful.

DRM_DDE The DDE rendering mechanism is intended for applications in which drag/drop actions will be used to set up DDE links between windows. The DDE then proceeds according to standard DDE protocols.

Further rendering mechanisms may be devised and documented for use by a particular user's applications. The creation and use of rendering mechanisms is discussed in 8.4, "Using Rendering Mechanisms" on page 187.

8.2 Data Structures Used in Drag/Drop

Three structures contain the data that travels with an item while it is being dragged; these are the DRAGINFO, DRAGITEM and DRAGIMAGE structures. A further structure, the DRAGTRANSFER structure, is used to transfer information between source and target windows after a drop has occurred.

Details of the fields within these structures can be found in the *IBM OS/2 Version 2.0 Presentation Manager Reference*, and full descriptions will not be given here. However, the following sections describe in general terms the kind of data the structures contain, and particularly certain critical fields.

8.2.1 The DRAGINFO Structure

The DRAGINFO structure contains information about the overall drag operation, which may consist of one or more dragitems. Information in the DRAGINFO structure determines the source and type of the drag operation, and provides pointers to one or more DRAGITEM structures which identify individual dragitems.

The handle of the source window for the drag operation is contained in the DRAGINFO structure. This handle enables a target window which receives the structure to initiate a conversation with the source window if necessary, in order to exchange information.

The other item of note in the DRAGINFO structure is a field which identifies the type of drag operation that the user has selected. For example, a value of DO_COPY means that the user is holding the Ctrl key down, which by convention means that a copy operation is required. The DO_DEFAULT value means that a default drag operation is to be used because the user is not holding down any modifier key.

The DRAGINFO structure contains a counter that specifies how many dragitems are involved in the current operation. This counter is then used to access an array of pointers, also contained within the DRAGINFO structure, which reference individual DRAGITEM structures for each dragitem.

Note that the DRAGINFO structure must be accessible not only to the source window that sets it up in the first place, but also to any potential target windows. Since these windows may not be owned by the same process, the DRAGINFO structure must be allocated in shared memory. In order that the structure be correctly allocated and easily accessible by the system in order to provide it to potential target windows, OS/2 allocates the DRAGINFO structure on the application's behalf, using the **DrgAllocDraginfo()** function. This function is called by the source window when it is notified of a drag operation by receiving a WM_BEGINDRAG message.

For Workplace Shell objects, the Workplace Shell handles the allocation and initialization of the DRAGINFO structure. The object itself is not required to take any action with respect to this structure.

8.2.2 The DRAGITEM Structure

The DRAGITEM structure contains information about an individual dragitem. A drag operation may include one or more dragitems, and a separate DRAGITEM structure is used for each. The number of dragitems, and an array of pointers to the DRAGITEM structures, is contained in the DRAGINFO structure. In conjunction with the information contained in the DRAGINFO structure, the DRAGITEM structure provides the information required by a potential or actual target window, to determine whether a drop operation is valid for the dragitem.

Several of the fields in the DRAGITEM structure are defined as being of type HSTR. These fields refer to ordinary null-terminated character strings that are given string handles by Presentation Manager when the **DrgAddStrHandle()** function is called. It is the string handles that are stored in the DRAGITEM structure; the strings themselves are stored by Presentation Manager and may be accessed by any other process that has access to the string handle, using the **DrgQueryStrName()** function.

For Presentation Manager windows, the DRAGITEM structure is normally allocated by the source of the drag/drop operation as a local variable, since it only persists for the duration of the operation. For Workplace Shell objects, the structure is allocated by the Workplace Shell and a pointer to it is passed to the object by the Workplace Shell when it invokes the object's *_wpFormatDragItem* method when the drag is initiated.

A number of fields in the DRAGITEM structure are of primary importance; these fields are described in the following sections.

8.2.2.1 ulItemID

This field contains a value provided by the source window, which uniquely identifies the dragitem. For example, the value might be a listbox index value, a customer number, or any other value that is unique and meaningful to the source window. The reason for having this identification is that later in the drag/drop processing, the target window may need to ask the source window for more information about the dragged item. The identifier can then be used to identify the item concerned.

8.2.2.2 hstrType and hstrRMF

These values refer to character strings containing details of the type of data represented by the dragitem, and the rendering mechanisms and formats that the source is able support for the item. The types correspond to the *file type* extended attribute, and are identified by names of the form DRT_xxxx; for example, DRT_TEXT for plain text, or DRT_BITMAP for bitmap data.

The rendering mechanism is specified in the *hstrRMF* field, and may refer to any of the standard mechanisms described in 8.1.3, "Rendering Mechanisms" on page 173, identified by the names DRM_PRINT, DRM_DISCARD, DRM_OS2FILE and DRM_DDE, or to any user-defined rendering mechanism for which a similar name should be defined. More than one rendering mechanism can be specified; for example, a program that allows the dragging of files may allow the file to be moved or copied to another directory, or to be printed by being dropped on a printer object. In this case the program would include the names of both the Print and OS/2 File rendering mechanisms in its hstrRMF string, allowing the target window to decide which is more suitable.

The format specifications, which are also contained in the *hstrRMF* field, inform a target window of the data formats supported by the dragitem for each of its supported rendering mechanisms. Data format names use the convention DRF_xxxx.

To illustrate the use of format specifications and rendering mechanisms, consider a spreadsheet program that allows the user to drag an icon representing a particular spreadsheet; the user may choose to drag the data into a word-processor, into another spreadsheet, or onto a printer object for printing. For dragging the file to another spreadsheet or to a word-processing document, the DRM_OS2FILE rendering mechanism is appropriate but for dragging to a printer object, the DRM_PRINT mechanism is required. In the case where the target is the printer or the word-processing document, clearly the required format for the data is text, but in the case of a drag to another spreadsheet it would be more convenient to have the numerical data and the cell relationships transferred too, so a different data format should be used, perhaps SYLK.

The dragitem therefore needs to indicate that it supports the following rendering mechanism/data format combinations:

- DRM_OS2FILE with DRF_TEXT
- DRM_PRINT with DRF_TEXT
- DRM_OS2FILE with DRF_SYLK.

The *hstrRMF* string provides a syntax for defining this in a fairly straightforward way. Complete details are given in the *OS/2 2.0 Programming Guide Volume II* but, for the above example, the *hstrRMF* string is as follows:

```
<DRM_OS2FILE,DRF_TEXT>,<DRM_PRINT,DRF_TEXT>,<DRM_OS2FILE,DRF_SYLK>
```

This can be expressed slightly more concisely as:

```
(DRM_OS2FILE,DRM_PRINT)x(DRF_TEXT),<DRM_OS2FILE,DRF_SYLK>
```

Here the first two bracketed items, connected with an "x," indicate that all possible pairs made up of one from the first bracket and one from the second, are implied. This notation is very useful in more complex examples, where it can save the programmer from having to enumerate all possible combinations in the string.

8.2.2.3 hstrContainerName, hstrSourceName, hstrTargetName

The meaning of these three fields depends on the rendering mechanism to be used; with some rendering mechanisms, certain fields are not needed. They apply most directly to the DRM_OS2FILE mechanism where they are used to define the source directory, source file name, and a suggested target filename (which may be overridden by the target window if it so chooses).

8.2.3 The DRAGIMAGE Structure

This structure, as its name suggests, contains information about the actual image to be displayed on the screen as the user performs the drag operation. In this structure, the source window specifies the icon or bitmap to be used, whether it is to be rescaled, and the coordinates of the hot spot.

8.2.4 The DRAGTRANSFER Structure

This structure is passed with a DM_RENDER message, from the target to the source window, after a drop has occurred. It allows the target window to inform the source of several important things. For example, where the source supports several different rendering mechanisms and/or formats, the target can specify which of these it wishes to use. Similarly, if the source supports both copy and move operations, the target can specify which it will use by means of the *usOperation* field of this structure.

Another important field is *hstrRenderToName*. This tells the source window where to place the data, so that the target will know where to find it. The precise interpretation of this depends on the rendering mechanism; for example, in the case of the DRM_OS2FILE rendering mechanism, it contains the fully qualified name that the file is to be given at its destination. Where the transfer of information between source and target window is a simple memory transfer, this field may be used to contain the name of a named shared memory object into which the source is to place the data.

8.3 Using Direct Manipulation

The following sections use an example to illustrate the way in which direct manipulation can be used within a Presentation Manager application or a Workplace Shell object. The example consists of a *Customer* program which reads and displays customer details. Each customer is displayed as an object in a container window.

The other component of the example is a *Telephone* program, which accepts customer information from the *Customer* program via drag/drop, and automatically dials the customer's telephone number. The *Telephone* program communicates with the *Customer* program using a private rendering mechanism defined by the application. This rendering mechanism uses shared memory, and is identified by the name DRM_SHAREMEM. The rendering mechanism is explained in detail in 8.4.2, "Implementing a Private Rendering Mechanism" on page 189.

The example uses Presentation Manager windows as both the source and target for the drag/drop operation, since this enables a description of the complete set of steps required to complete the operation. For Workplace Shell objects, certain steps are handled automatically by the Workplace Shell itself, and a Workplace Shell object class is therefore not required to carry out these steps. Where a particular step is automated by the Workplace Shell, this is noted in the discussion.

8.3.1 Initiating a Drag Operation

A drag operation is initiated by the source window or object. When the user starts the drag operation by pressing and holding down mouse button 2, Presentation Manager passes a WM_BEGINDRAG message to the window or object that is currently under the mouse pointer. In the case of a Presentation Manager window, the window procedure for that window may process the WM_BEGINDRAG message in order to initialize the DRAGINFO and DRAGITEM structures, and start the drag operation. A Workplace Shell object is notified of a drag initiation by the Workplace Shell itself, which invokes the object's *_wpFormatDragItem* method.

The initialization of a drag operation from a container window is shown in Figure 81 on page 179.

```
PCONTRECORD  pCRec;
PCNRDRAGINIT pcnrInit;
PDRAGINFO    pDInfo;
DRAGITEM     DItem;
DRAGIMAGE    DImage;

APIRET       rc;
  :
  :
case WM_CONTROL:
     switch (SHORT2FROMMP(mp1))
            {
            case CN_INITDRAG:
                 pcnrInit =                    /* Get container data    */
                    (PCNRDRAGINIT)mp2;
                 pCRec = (PCONTRECORD)pcnrInit->pRecord;

                 if (pCRec == NULL)            /* If no item selected   */
                    return(0);                 /* Return                */

                 pDInfo = DrgAllocDraginfo(1); /* Allocate DRAGINFO     */

                 DItem.hwndItem = hWnd;        /* Initialize DRAGITEM   */
                 DItem.ulItemID = (ULONG)pCRec;
                 DItem.hstrType =
                    DrgAddStrHandle("DRT_CUSTOMER");
                 DItem.hstrRMF =
                    DrgAddStrHandle("(DRM_SHAREMEM,DRM_PRINT)x(DRF_TEXT)");
                 DItem.fsControl = 0;
                 DItem.fsSupportedOps = DO_COPYABLE | DO_MOVEABLE;

                 rc = DrgSetDragItem(pDInfo,   /* Set item in DRAGINFO  */
                        &DItem,                /* Pointer to DRAGITEM   */
                        sizeof(DItem),         /* Size of DRAGITEM      */
                        0);                    /* Index of DRAGITEM     */

                 DImage.cb = sizeof(DRAGIMAGE); /* Initialize DRAGIMAGE  */
                 DImage.cptl = 0;              /* Not a polygon         */
                 DImage.hImage = hPostIcon;    /* Icon handle for drag  */
                 DImage.fl = DRG_ICON;         /* Dragging an icon      */
                 DImage.cxOffset = 0;          /* No hotspot            */
                 DImage.cyOffset = 0;

                 hDrop = DrgDrag(hWnd,         /* Initiate drag         */
                           pDInfo,             /* DRAGINFO structure    */
                           (PDRAGIMAGE)&DImage, /* DRAGIMAGE structure  */
                           1,                  /* Only one DRAGIMAGE    */
                           VK_ENDDRAG,         /* End of drag indicator */
                           NULL);              /* Reserved              */

                 DrgFreeDragInfo(pDInfo);      /* Free DRAGINFO struct  */
                 break;
```

Figure 81. Drag Initiation From a Container Window. Another window class would perform these operations in response to a WM_BEGINDRAG message, rather than a WM_CONTROL message with the CN_INITDRAG indicator.

The code shown in Figure 81 would form part of the window procedure for the owner of the container control, since it is this window that would receive the WM_CONTROL message from the container.

8.3.1.1 Initializing Data Structures

When a WM_CONTROL message is received from a container window, a pointer to a CNRDRAGINIT structure is passed in the second parameter to the WM_CONTROL message. This structure contains a pointer to the item within the container that the user is attempting to drag. If this pointer is NULL, the user has attempted a drag operation while no item in the container was selected. In the current example, the drag operation is ignored and control is immediately returned to Presentation Manager.

The source window then allocates a DRAGINFO structure using the **DrgAllocDraginfo()** function. The DRAGITEM structure is initialized with the appropriate values, and its pointer is set in the DRAGINFO structure using the **DrgSetDragItem()** function. All interaction with the DRAGINFO structure is performed using Presentation Manager functions, avoiding the necessity for the source window procedure to address the DRAGINFO structure directly.

The DRAGIMAGE structure is then initialized with the information relating to the icon that will be displayed under the mouse pointer during the drag operation.

For a Workplace Shell object, the Workplace Shell itself performs the initialization of the DRAGINFO structure. The object may perform its own initialization of the DRAGITEM structure during processing of the *_wpFormatDragItem* method, if class-specific processing is required. For example, if the object class implements a private rendering mechanism, the appropriate information must be entered into the correct fields in the DRAGITEM structure as part of the *_wpFormatDragItem* method.

Note that a Workplace Shell object need not allocate the DRAGITEM structure, since the structure is already allocated by the Workplace Shell, and a pointer to the structure is passed to the *_wpFormatDragItem* method upon invocation.

8.3.1.2 DrgDrag() Processing

Once all data structures are allocated and initialized, the drag operation is initiated using the **DrgDrag()** function. This function is synchronous; it does not return control to the source window procedure until the key or mouse button specified in the fifth parameter (VK_ENDDRAG in the example above) is detected, and any synchronous message passing has been completed.

At this point, the **DrgDrag()** function returns a window handle. If the dragitem was dropped over a window or object that was able to accept the item, the window handle of the target window is returned. If a drop occurred over an object that was unable to accept the item, a NULL window handle is returned.

Upon return of control by the **DrgDrag()** function, the drag operation and the drop operation (if any) is complete, and the DRAGINFO structure can be released by the source window procedure.

A Workplace Shell object is not required to invoke the **DrgDrag()** function, since this is performed automatically by the Workplace Shell when the object completes the processing of its *_wpFormatDragItem* method.

8.3.1.3 Synchronous Message Processing During DrgDrag()

When the user drops the dragitem over another window or object that is able to accept the item, a DM_DROP message is passed to the target, which then processes the drop operation. Note that the target's DM_DROP message processing must complete and return control to Presentation Manager before the **DrgDrag()** function will return control to the source window procedure. Thus any processing that is performed by the target window during its processing of the DM_DROP message is synchronous.

The synchronous nature of this processing is necessary in order to ensure that the drop operation, and the accompanying transfer of information, is completed before the user performs any other operation. For this reason, it is recommended that any messages passed by the target to the source window during processing of the DM_DROP message should be passed synchronously using the **DrgSendTransferMsg()** function. This is a departure from the normal Presentation Manager guidelines, where messages are processed asynchronously, but is required in order to ensure data integrity.

A number of synchronous messages may be sent to the source window at the completion of a drop, prior to the **DrgDrag()** call returning control to the source window. For example, if the user drops an object on a Workplace Shell printer object with the DRM_PRINT rendering mechanism specified, the target object sends a DM_PRINTOBJECT message to the source window. This message contains sufficient information for the source window to direct a print data stream to the print queue represented by the printer object. The first parameter in the DM_PRINTOBJECT message contains a pointer to the DRAGITEM structure that identifies the item being dropped, and the second parameter contains the name of the print queue for the printer object.

An example of the way in which the source window procedure may process the DM_PRINTOBJECT message is shown in Figure 82.

```
case DM_PRINTOBJECT:
     WinMessageBox(HWND_DESKTOP,                /* Display message box   */
                   hWnd,                        /* Curr window is owner  */
                   "Printing customer details", /* Message box text      */
                   "Print Message Received",    /* Message box title     */
                   0,                           /* No identifier         */
                   MB_OK);                      /* Include okay button   */

     <Extract DRAGITEM pointer from mp1>
     <Extract print queue name from mp2>

     <Print item>

     break;
```

Figure 82. Receiving a DM_PRINTOBJECT Message

Note that the code that actually performs the printing operation has been omitted from Figure 82. Printing under OS/2 Version 2.0 and Presentation Manager is discussed in detail in *OS/2 Version 2.0 - Volume 5: Print Subsystem*, and examples are also provided in the PRTSAMP program included in the IBM Developer's Toolkit for OS/2 2.0.

8.3.2 Dragging Over a Window

While the dragitem is being dragged, Presentation Manager sends a succession of DM_DRAGOVER messages to the Presentation Manager window under the mouse pointer; one message is sent for every mouse movement. The DM_DRAGOVER message informs the target window that it is being dragged over, and allows it to access sufficient information to allow the window to decide whether it is able to accept a drop operation. The window procedure indicates this to Presentation Manager by the value that it returns in response to the DM_DRAGOVER message.

The information required by the window is contained in two data structures; the DRAGINFO structure, which is referenced by one of the parameters in the DM_DRAGOVER message, and the DRAGITEM structure, which can be accessed from the DRAGINFO structure. Since multiple items may be dragged at the same time, the DRAGINFO structure contains an array of pointers to DRAGITEM structures, one for each dragitem. The DRAGINFO structure maintains a count of the number of dragitems.

When a potential target is a Workplace Shell object, the Workplace Shell notifies the object that a dragitem is being dragged over it, by invoking the object's *_wpDragOver* method. The DRAGINFO structure is passed to the object as a parameter to this method. Processing of the *_wpDragOver* method is very similar to that described below for Presentation Manager window procedures. In normal circumstances, however, the *_wpDragOver* method is not overridden by an object class; the default processing supplied by the parent class is allowed to occur unless the object class supports private rendering mechanisms that must be explicitly checked against those supported by the dragitem.

An example of the way in which a window procedure may process the DM_DRAGOVER message is shown in Figure 83.

```
PDRAGITEM pDItem;                          /* Pointer to DRAGITEM   */
PDRAGINFO pDInfo;                          /* Pointer to DRAGINFO   */

case DM_DRAGOVER:
     pdinfo = (PDRAGINFO)mp1;              /* Get DRAGINFO pointer  */
     DrgAccessDraginfo(pdinfo);            /* Access DRAGINFO       */
     pditem = DrgQueryDragitemPtr(pdinfo,  /* Access DRAGITEM       */
                                  0);      /* Index to DRAGITEM     */
     if (!DrgVerifyRMF(pditem,             /* Check valid rendering */
                       "DRM_SHAREMEM",     /* mechanisms and data   */
                       "DRF_TEXT"))        /* formats               */
       {
       DrgFreeDraginfo(pdinfo);            /* Free DRAGINFO         */
       return(MPFROM2SHORT(DOR_DROP,       /* Return okay to drop   */
                           DO_COPY));      /* Copy operation valid  */
       }
     else
       {
      DrgFreeDraginfo(pdinfo);             /* Free DRAGINFO         */
      return(MPFROM2SHORT(DOR_NEVERDROP,   /* Drop not valid        */
                          0));             /* No valid operations   */
       }
     break;
```

Figure 83. Handling the DM_DRAGOVER Message

The code shown in Figure 83 is quite simple; the processing of the DM_DRAGOVER message is intended only to determine whether a drop operation is valid for the specified dragitem and the target window. First, access is gained to the DRAGINFO structure, which is referenced by the first parameter to the DM_DRGOVER message. The DRAGINFO structure is then used to access the DRAGITEM structure, by means of the **DrgQueryDragitemPtr()** function. The window procedure then has access to all the information needed to determine the validity of a drop operation.

In this particular case, the only type of dragitem that is acceptable to the target window is one that represents a customer object, using the specially defined DRM_SHAREMEM rendering mechanism. The window procedure therefore uses the **DrgVerifyRMF()** function to check whether the dragitem supports this rendering mechanism and the data type required by it.

According to the result returned by the **DrgVerifyRMF()** function, the window procedure returns either DOR_DROP, indicating that a drop is acceptable, or DOR_NEVERDROP, indicating that a drop is not acceptable and that there is no point in sending any more DM_DRAGOVER messages to this window. A number of other valid returns are possible for the DM_DRAGOVER message; these are documented in the *IBM OS/2 Version 2.0 Presentation Manager Reference.* Irrespective of the return code, the DRAGINFO structure is released.

If a window returns any return code other than DOR_DROP to this message, the icon seen by the user is automatically modified to show that a drop is not allowed, thereby providing instant visual feedback.

8.3.3 Dropping an Object

When the user drops a dragitem over a Presentation Manager window, the target receives a DM_DROP message. The window procedure for the target may process that message in order to handle the drop operation, and may either complete the operation or initiate a conversation with the source window or object in order to do so, typically by sending it a DM_RENDER message, which ultimately will result in the data being transferred.

When the user drops a dragitem over a Workplace Shell object, the Workplace Shell invokes that object's *_wpDrop* method. The processing of this method is very similar to that discussed below for the DM_DROP message. However, object classes that do not implement private rendering mechanisms need not override the *_wpDrop* method; the default processing provided by the parent class may be allowed to occur.

In the customer/phone dialler example, the only type of dragitem that the order program will accept is a customer object, which uses the application-defined DRM_SHAREMEM rendering mechanism. The correct data type and rendering mechanism is verified by the target window procedure during processing of the DM_DRAGOVER message, so there is no need for further checking when the DM_DROP is processed. Note however, that in a more sophisticated application, which supports multiple data types and rendering mechanisms, it may be necessary to perform more detailed checking.

```
#define XFERMEM "\\SHAREMEM\\DragXfer.mem"      /* Shared mem obj name   */

PVOID     pCust;                                /* Customer record ptr   */

PDRAGITEM pDItem;                               /* DRAGITEM struct ptr   */
PDRAGINFO pDInfo;                               /* DRAGINFO struct ptr   */
  :
case DM_DROP:
     pDInfo = (PDRAGINFO)mp1;                   /* Get DRAGINFO pointer  */
     DrgAccessDraginfo(pDInfo);                 /* Access DRAGINFO       */
     pDItem = DrgQueryDragitemPtr(pdinfo,       /* Access DRAGITEM       */
                                  0);           /* Index to DRAGITEM     */

     DosAllocSharedMem(&pCust,                  /* Allocate shared mem   */
                       XFERMEM,                 /* Named memory object   */
                       sizeof(CUSTOMER),        /* Size of memory object */
                       PAG_COMMIT |             /* Commit storage now    */
                       PAG_WRITE  |             /* Allow write access    */
                       PAG_READ);               /* Allow read access     */

     pdxfer = DrgAllocDragtransfer(1);          /* Allocate DRAGTRANSFER */
     pdxfer->cb = sizeof(DRAGTRANSFER);         /* Init DRAGTRANSFER     */
     pdxfer->hwndClient = hWnd;
     pdxfer->pditem = pDItem;
     pdxfer->hstrSelectedRMF =
            DrgAddStrHandle("<DRM_CUSTOMER,DRF_TEXT>");
     pdxfer->hstrRenderToName =
            DrgAddStrHandle(XFERMEM);
     pdxfer->ulTargetInfo = 0;
     pdxfer->usOperation = DO_COPY;

     rc=DrgSendTransferMsg(pDInfo->hwndSource,  /* Send msg to source    */
                           DM_RENDER,           /* DM_RENDER message     */
                           (MPARAM)pdxfer,      /* DRAGTRANSFER pointer  */
                           NULL);
     if (rc == TRUE)                            /* If rendered okay      */
        {
        strcpy(msgtext, "Dialling number");     /* Build message text    */
        strncat(msgtext,
                pxfercust->phone,
                30);
        WinMessageBox(HWND_DESKTOP,             /* Display message box   */
                      hWnd,                     /* Curr window is owner  */
                      msgtext,                  /* Message text          */
                      "Telephone Dialler",      /* Message title         */
                      0,                        /* No identifier         */
                      MB_OK);                   /* Include okay button   */

        PhoneDial(pxfercust->phone);            /* Dial number           */
        }

     DrgFreeDragInfo(pdinfo);                   /* Release all data      */
     DrgFreeDragtransfer(pdxfer);               /* structures            */
     DosFreeMem((PVOID)pxfercust);
     break;
```

Figure 84. Handling the DM_DROP Message

In Figure 84, access must first be gained to the DRAGINFO and DRAGITEM structures. This is achieved in a similar manner to that already described for the DM_DRAGOVER message. Having gained access to these structures, a named shared memory object is then allocated, into which the source window will be asked to place the customer details.

A DRAGTRANSFER structure is then allocated, in which information about the target's requirements can be passed to the source window. This structure is similar to the DRAGINFO structure, in that it must be accessible from multiple processes. It is therefore allocated using the **DrgAllocDragtransfer()** function, ensuring that the structure will be accessible to the source window, which may be in another process and therefore not have direct access to the target's private data areas.

There are several important fields in this structure. The target window procedure places a pointer to the DRAGITEM structure into the *pditem* field, thereby enabling the source to identify which item has been dropped. The *hstrSelectedRMF* field is used to identify which rendering mechanism and data format is to be used for this target, from the selection offered by the source in the DRAGITEM structure. The *hstrRenderToName* field is used in the DRM_SHAREMEM rendering mechanism to pass the name of the shared memory object to the source window.

Once this structure has been completed with the necessary information, it is sent to the source window as part of a DM_RENDER message. This message is passed to the source window using the **DrgSendTransferMsg()** function. This function should be used for drag/drop operations in preference to the **WinSendMsg()** function since, for a DM_RENDER message, it also grants access to the DRAGTRANSFER structure for the process that owns the window to which the message is being sent.

In processing the DM_RENDER message, the source window copies the customer details into the shared memory so that when **DrgSendTransferMsg()** returns, the target window procedure may extract the data it needs. A detailed explanation of the source window's processing of a DM_RENDER message is given in 8.3.4, "Transferring Information."

Upon completion of the information transfer, the entire drag/drop operation is complete and the data structures allocated during the operation may be released. For the DRAGINFO and DRAGTRANSFER structures, this must be carried out using the **DrgFreeDraginfo()** and **DrgFreeDragtransfer()** functions.

8.3.4 Transferring Information

As explained in 8.3.3, "Dropping an Object" on page 183, a target window may send a DM_RENDER message to the source when it receives a DM_DROP message from Presentation Manager. Similarly, a Workplace Shell object may send the same message when its *_wpDrop* method is invoked by the Workplace Shell. This message is normally sent to the source when the target requires the assistance of the source in completing the transfer of data as part of the drop operation.

The source window processes this message in its window procedure, according to the rendering mechanism requested by the target. If the source is a Workplace Shell object, the Workplace Shell will directly invoke the object's *_wpRender* method to perform the same function. In most cases, however, an

object does not need to override the *_wpRender* method unless it wishes to implement a private rendering mechanism.

The DM_RENDER processing from the *Customer* program is shown in Figure 85.

```
PDRAGITEM     pDItem;                          /* DRAGITEM pointer      */
PDRAGINFO     pDInfo;                          /* DRAGINFO pointer      */
PDRAGTRANSFER pDXfer;                          /* DRAGTRANSFER pointer  */
PCONTRECORD   pCRec;                           /* Container record ptr  */
PCUSTOMER     pCust,                           /* Customer record ptrs  */
              pXferData;
CHAR          xfermem[100];                    /* Memory name buffer    */

HWND          hContainer;                      /* Container handle      */
  :
  :
case DM_RENDER:
     pDXfer = (PDRAGTRANSFER)mp1;              /* Get DRAGTRANSFER ptr  */
     pDItem = pDxfer->pditem;                  /* Get DRAGITEM ptr      */
     pCRec = pditem->ulItemID;                 /* Get container rec ptr */
     pCust = pCRec->cust;                      /* Get customer rec ptr  */

     DrgQueryStrName(pDXfer->hstrRenderToName, /* Get mem object name   */
                     100,                      /* Size of buffer        */
                     xfermem);                 /* Buffer                */

     DosGetNamedSharedMem((PPVOID)&pXferData,  /* Get shared mem object */
                          xfermem,             /* Name of mem object    */
                          PAG_WRITE |          /* Allow write access    */
                          PAG_READ);           /* Allow read access     */
     memcpy(pCust,                             /* Copy customer record  */
            pXferData,                         /* to shared mem object  */
            sizeof(CUSTOMER));                 /* No. of bytes to copy  */

     DosFreeMem((PVOID)pCust);                 /* Free shared mem obj   */

     if (pDXfer->usOperation == DO_MOVE)       /* If move operation     */
        {
        hContainer = WinWindowFromID(hWnd,     /* Get container window  */
                        CONTAINER);            /* handle                */
        RemoveCustomer(hContainer,             /* Remove record from    */
                       pCRec);                 /* container             */
        }
     return((MRESULT)TRUE);                    /* Return TRUE           */
     break;
```

Figure 85. Handling the DM_RENDER Message

The first parameter to the DM_RENDER message contains a pointer to the DRAGTRANSFER structure, which in turn contains a pointer to the DRAGITEM structure in its *pditem* field. For a Workplace Shell object, a pointer to the DRAGTRANSFER structure is passed as a parameter to the *_wpRender* method.

In the DRM_SHAREMEM rendering mechanism, the *ulItemID* field in the DRAGITEM structure is used to hold a pointer to the customer container record (of type CONTRECORD), in whch the *cust* field is a CUSTOMER structure containing details of the customer object which was dragged.

Next, the name of the shared memory object previously allocated by the target window is retrieved from the *hstrRenderToName* field of the DRAGTRANSFER structure. This name is used to obtain access to the shared memory object. The customer details are copied into this memory object, after which the memory object is freed.

The operation code in the DRAGTRANSFER structure is then checked to establish whether the target requires a copy or a move operation. If a move was requested, the source program deletes the customer record from the contaiiner by calling an application subroutine named *RemoveCustomer()*.

The window procedure then returns the value TRUE, indicating that the data was successfully rendered. This value is returned to the target window procedure that issued the **DrgSendTransferMsg()** call. At this point, the target window procedure has access to all information required to complete the drop operation, and may do so without further communication.

At the completion of the rendering procedure, the source may pass a DM_RENDERCOMPLETE message to the target, allowing the target to release any resources still outstanding. A Presentation Manager window may process this message in its window procedure, while a Workplace Shell object is notified of the event by the Workplace Shell, which invokes the object's *_wpRenderComplete* method. This is usually only required in cases where complex private rendering mechanisms involve multiple transfers. It is not used in the above examples.

8.4 Using Rendering Mechanisms

The rendering mechanism is essentially a protocol that determines the contents of several fields in the DRAGITEM structure. These fields are:

- *ulItemID*, which contains an application-specific value uniquely identifying the item being dragged.
- *hstrType*, which contains a handle to a string defining the data type of the dragitem.
- *hstrRMF*, which contains a handle to a string containing the names of all rendering mechanisms supported by the dragitem, and the data formats supported by those rendering mechanisms.
- *hstrContainerName*, *hstrSourceName* and *hstrTargetName*, which contain handles to strings used by the DRM_OS2FILE rendering mechanism, and may be used by private rendering mechanisms to contain string data.

The content of the *hstrRMF* field should obey a set of syntactical rules that are explained in the *OS/2 2.0 Programming Guide Volume II*. Other fields in the DRAGITEM structure may also be used by particular rendering mechanisms; their use is dependent upon the individual rendering mechanism in use at the time. Applications may use one of the standard rendering mechanisms DRM_PRINT, DRM_DISCARD, DRM_OS2FILE or DRM_DDE, or may define their own rendering mechanisms to support dragging and dropping of particular dragitems.

8.4.1 Standard Rendering Mechanisms

The following sections describe the use of two of the standard rendering mechanisms, DRM_PRINT and DRM_OS2FILE. These mechanisms can be used by Presentation Manager applications to interact with other applications and/or Workplace Shell objects.

8.4.1.1 DRM_PRINT

For Presentation Manager applications running on the Workplace Shell desktop under OS/2 Version 2.0, it may be desirable to allow the user to print from the program by dragging the relevant item, such as a customer record, onto a Workplace Shell printer object. Since all Workplace Shell printer objects are written to understand the DRM_PRINT rendering mechanism, a Presentation Manager may provide such function simply by adhering to this mechanism.

With the DRM_PRINT rendering mechanism, responsibility for actually carrying out the printing rests within the source window. The source window must:

1. Detect the fact that a drag is being initiated by the user
2. Allocate and fill the DRAGINFO and DRAGITEM structures
3. Start the drag operation using the **DrgDrag()** function
4. Process the DM_PRINTOBJECT message which is returned by the target printer object.

The first three steps are handled in exactly the same way as illustrated in Figure 81 on page 179. Note that a view of a Workplace Shell object need not explicitly handle Presentation Manager messages in the window procedures for its views. When the user initiates a drag from within a Workplace Shell object such as a folder or work area, the object is notified by the Workplace Shell, which invokes the object's *_wpFormatDragItem* method. This method is processed in an identical manner to that shown for the window procedure in Figure 81.

The final step is handled by processing the DM_PRINTOBJECT message in the source window procedure. A simple example of such processing is shown in Figure 82 on page 181.

8.4.1.2 DRM_DISCARD

The DRM_DISCARD rendering mechanism operates in a similar fashion to the DRM_PRINT mechanism, and is intended for use by applications which create their own equivalent to the Workplace Shell *Shredder* object. In this rendering mechanism, the target passes a DM_DISCARDOBJECT message to the source, which may either accept responsibility for the discard operation, abort the operation, or allow the system to perform the operation.

Note that the system may only discard objects which are capable of being rendered with the DRM_OS2FILE rendering mechanism; that is, program files and data files. Other objects not based upon files must be explicitly discarded by the source.

When a Workplace Shell object is dropped on the *Shredder* object, the Workplace Shell intercepts the DM_DISCARDOBJECT message and invokes the source object's *wpDelete* method.

8.4.1.3 DRM_OS2FILE

The DRM_OS2FILE rendering mechanism is designed to support moving and copying file objects between containers. This rendering mechanism is described in detail in the *OS/2 2.0 Programming Guide Volume II*, and an extensive example is provided in the IBM Developer's Toolkit for OS/2 2.0. The details of programming for the DRM_OS2FILE mechanism will therefore not be described further in this document.

Certain fields in the DRAGITEM structure are designed specifically for this rendering mechanism; the *hstrContainer*, *hstrSourceName* and *hstrTargetName* fields are ideally suited to holding the source directory name, source file name, and fully qualified target file name respectively. This is how these fields are used by the DRM_OS2FILE rendering mechanism.

An alternative, and even more straightforward way to implement this rendering mechanism, is to use the **DrgDragFiles()** function. This function automatically allocates and fills the required data structures for the source window, avoiding the need for the application to perform these functions and reducing the risk of error.

8.4.2 Implementing a Private Rendering Mechanism

The *OS/2 2.0 Programming Guide Volume II* gives some advice on use of the various messages available to implement a private rendering mechanism, and also some guidelines on how such a rendering mechanism should be documented. This section illustrates the implementation of a simple rendering mechanism, by explaining the definition of the DRM_SHAREMEM rendering mechanism used by the examples earlier in this chapter.

A rendering mechanism is necessary to pass the customer record data used in the examples, since the CUSTOMER structure that contains this data is too large to be contained within the DRAGITEM structure. It is therefore necessary, after a drop has occurred, for the source program to make the relevant data available to the target, in a format which is understood by and accessible to both the source and the target. In the examples, a named shared memory object is used to transfer the data; hence the name DRM_SHAREMEM used for the rendering mechanism.

The DRM_SHAREMEM rendering mechanism operates as follows:

- The source window stores a pointer to the customer record being dragged in the *ulItemID* field of the DRAGITEM structure. This field is defined as ULONG, but it can be used in any way that is meaningful to the source window to identify the item being dragged. A pointer to the customer record is a convenient way to do this.
- The target window, on receiving a DM_DROP message, allocates a named shared memory object with a name of its choice. It then sends a DM_RENDER message to the source window, passing the name of the memory object in the *hstrRenderToName* field of the DRAGTRANSFER structure, and indicating whether it requires a copy (DO_COPY) or a move (DO_MOVE) to take place, using the usOperation field of the DRAGTRANSFER structure.
- When the source window receives the DM_RENDER message, it obtains access to the shared memory object and places the customer record in that object. The source window knows which customer record to copy, since the

DRAGITEM structure, which includes a pointer to the customer record, is passed along with the DRAGTRANSFER structure.

Finally, if a move operation was requested by the target, the source window deletes the customer record from its own data.

- On receiving a TRUE return code from the DM_RENDER message, indicating that the data was successfully rendered, the target window copies the data out of the shared memory object, and uses it in whatever way it chooses.

It should be stressed that this is a very simple rendering mechanism. However, it illustrates the general structure of such mechanisms, and their impact on the contents of fields in the DRAGITEM and DRAGTRANSFER structures.

8.5 Summary

Direct manipulation is likely to become considerably more important to application designers than it has been in previous releases of OS/2, because of its central role in the object-oriented user interface provided by the Workplace Shell. Even applications that are not implemented as Workplace Shell objects should provide, so far as is practical, a similar style of interface. Direct manipulation forms an essential part of such an interface.

The programming facilities for direct manipulation in OS/2 V2.0 are essentially the same as those introduced in OS/2 Version 1.3, and consist of a set of message classes, functions and data structures, along with defined protocols known as rendering mechanisms, which define standard techniques for using these facilities to pass different types of information between diverse applications, and between user-developed applications and Workplace Shell objects such as printers and the shredder.

User-defined rendering mechanisms may also be defined for specific purposes that are not covered by the standard ones. The *OS/2 2.0 Programming Guide Volume II* gives guidance on this and on how such rendering mechanisms should be documented.

Chapter 9. Presentation Manager Resources

The definition and use of Presentation Manager resources by applications was mentioned in Chapter 4, "The Presentation Manager Application Model." The use of such resources greatly simplifies the task of the application developer in creating windows, menu bars, etc., and provides a powerful tool for the externalization of the user interface properties of an application object, thereby enabling easier modification of these properties during development or maintenance of the application. This chapter will describe the definition of resources, and the ways in which resources may be used within a Presentation Manager application.

9.1 Types of Resources

A number of different types of resources may be defined for use by Presentation Manager applications. These include text items such as menu bars and window templates, and graphical items such as graphics fonts, icons and bitmaps. Textual items are defined in the **resource script file**, which is described in 9.2, "Resource Script File" on page 198. Non-textual items are defined and saved in other files, and are referenced by statements in the resource script file. The various types of Presentation Manager resource are described in the following sections.

9.1.1 Fonts

A font is a set of alphanumeric characters and other symbols. Fonts may be designed interactively using the Font Editor application provided as part of the *IBM Developer's Toolkit for OS/2 2.0*. Once a font has been designed, the Font Editor saves the font in a disk file with an extension .FNT. This font file is referenced from the resource script file using the FONT keyword:

```
FONT     123     MYFONT.FNT
```

The integer following the FONT keyword is used to identify the font resource. A symbolic name *cannot* be used to define a font resource.

A font file must be link-edited along with a resource script file containing a FONT keyword referencing the .FNT file, and with the FONTS.OBJ and FONTS.DEF files provided with the *OS/2 Programmer's Toolkit*, to produce a file with an extension of .FON. Although a font cannot be stored in a dynamic link library, the .FON file may be installed on the system by the user or installed explicitly by an application. The font is then usable by any application in the system. Alternatively, an application developer may choose not to install the font, but merely to access it from a particular application using the **GpiLoadFonts()** function.

9.1.2 Icons, Pointers and Bitmaps

As already mentioned, an icon is a graphical representation of an object on the screen. For the purposes of discussion, icons, pointers and bitmaps will be grouped together; a pointer is a graphical image that is associated with a pointing device such as a mouse, and which moves on the screen as the pointing device is moved by the user, whereas a bitmap is a graphical image that typically is used to represent a general item such as a logo. Icons, pointers and bitmaps may be designed interactively using the Icon Editor application

supplied as part of the OS/2 Version 2.0 product. Depending on which resource is being created, the Icon Editor saves the resulting icon, pointer or bitmap in a file with an extension of ICO, PTR or BMP. These files are then referenced from the resource script file using the ICON, POINTER or BITMAP keywords:

```
ICON     MAIN     APPLIC.ICO
POINTER  DRAW     PENCIL.PTR
BITMAP   INTRO    LOGO.BMP
```

The keyword is followed in each case by a resource identifier, which is a symbolic name used by the application to identify the resource. For an icon, the identifier is used as a parameter to the **WinCreateWindow()** and **WinCreateStdWindow()** calls, and identifies the icon resource to be used when the FCF_ICON attribute is specified for the frame window. In all cases, the symbolic name must be defined as an integer constant using a *#define* statement.

For a pointer or bitmap, the identifier is used as a parameter to the **WinLoadPointer()** or **GpiLoadBitmap()** functions, which load the resource into memory. **WinLoadPointer()** returns a pointer to the resource in memory, which may then be used as a parameter to the **WinSetPointer()** function, in order to set the desktop pointer to that resource.

A pointer may be set to one of the system-defined pointer styles (such as an arrow or hourglass) using the **WinSetPointer()** function, by obtaining the handle of the required system pointer using the **WinQuerySysPointer()** function as follows:

```
rc = WinSetPointer(HWND_DESKTOP,
                   WinQuerySysPointer(HWND_DESKTOP,
                                      SPTR_WAIT,
                                      FALSE));
```

This call will set the pointer for the desktop to the hourglass pointer (indicated by the symbolic name SPTR_WAIT). The handle of the hourglass pointer is returned by the **WinQuerySysPointer()** call. The symbolic names of the various system pointers are described along with the **WinQuerySysPointer()** function in the *IBM OS/2 Version 2.0 Presentation Manager Reference*.

A bitmap is drawn within a window on the screen using the **WinDrawBitmap()** function. The pointer to the bitmap, returned by **GpiLoadBitmap()** is passed as a parameter to **WinDrawBitmap()** in order to identify the resource.

9.1.3 Menu Bars and Pulldown Menus

Menu bars and their associated pulldown menus are defined within the resource script file, using the MENU and SUBMENU keywords. A sample menu bar and pulldown menu definition is shown in Figure 86 on page 193.

```
MENU MAIN      PRELOAD
BEGIN
     SUBMENU "~File",  MI_FILE, MIS_TEXT
     BEGIN
          MENUITEM "~New...",  MI_NEW, MIS_TEXT
          MENUITEM "~Open",    MI_OPEN, MIS_TEXT
          MENUITEM "~Save",    MI_SAVE, MIS_TEXT
          MENUITEM "Save ~as", MI_SAVEAS, MIS_TEXT
     END
     SUBMENU "~Edit", MI_EDIT, MIS_TEXT
     BEGIN
          MENUITEM "Cu~t",     MI_CUT, MIS_TEXT
          MENUITEM "~Copy",    MI_COPY, MIS_TEXT
          MENUITEM "~Paste",   MI_PASTE, MIS_TEXT
     END
     SUBMENU "~Window",        MI_WINDOW, MIS_TEXT
     BEGIN
          MENUITEM "~Tile",    MI_TILE, MIS_TEXT
          MENUITEM "~Cascade", MI_CASC, MIS_TEXT
     END
     MENUITEM "E~xit",         MI_EXIT, MIS_TEXT
     MENUITEM "~Help",         MI_HELP, MIS_HELP |
                                       MIS_BUTTONSEPARATOR
END
```

Figure 86. Menu Bar Resource Definition

The symbolic name (MAIN in the example above) identifies the resource. This name is passed as a parameter to the **WinCreateWindow()** and **WinCreateStdWindow()** functions and identifies the menu bar resource when the FCF_MENU style frame control flag is specified for the frame window. The PRELOAD option specifies that the resource will be incorporated into the application's main .EXE file, and is to be loaded immediately into memory, rather than being loaded when called by the application.

The SUBMENU statement defines a menu bar entry that will be associated with a pulldown menu. MENUITEM statements that are enclosed within the BEGIN and END markers of a SUBMENU statement define the pulldown menu items. MENUITEM statements that are not enclosed within the bounds of a SUBMENU statement define menu bar entries that do not have an associated pulldown menu.

The text strings within quotation marks define the text for each menu bar or pulldown menu item. The symbolic name following the text identifies the value (placed in the first message parameter) of the WM_COMMAND message generated when the item is selected. The symbolic names following the message identifier define the style of the item. In the example above, all items are simple text items and are defined with the style MIS_TEXT. The sole exception is the final "Help" menu bar item, which is defined with the style attributes of MIS_HELP, which causes the item to generate a WM_HELP message rather than a WM_COMMAND message, and MIS_BUTTONSEPARATOR, which causes the item to be displayed on the right-hand side of the menu bar separated by a vertical bar, in accordance with SAA CUA guidelines. The various item styles and attributes are documented in the *IBM OS/2 Version 2.0 Presentation Manager Reference*.

When groups of items within a single pulldown menu are logically separate, they should be visually separated by a horizontal bar within the pulldown menu. This may be achieved using the SEPARATOR keyword in the MENUITEM statement, as follows:

```
MENUITEM  SEPARATOR
```

The use of a separator bar in pulldown menus is particularly important when the pulldown menu is used to display a list of entries, comprised of multiple sets of mutually exclusive options, from which the user must select one option from each set. In such a case, the separator bar is used to group the items within each set, and to visually separate the sets from one another.

As already mentioned, menu bar resources are typically incorporated into a window by specifying their resource identifier in a **WinCreateWindow()** or **WinCreateStdWindow()** call, with the FCF_MENU frame creation flag set for the frame window. A submenu within a menu bar resource may also be dynamically created using the **WinCreateMenu()** function, which is described in 11.2, "The Menu Bar" on page 241.

9.1.3.1 Mnemonics

Mnemonics may be specified for menu bar and pulldown menu items. A mnemonic is a key which, when combined with the F10 key, results in selection of the item. The character for the mnemonic must be part of the text for the item. For example, the conventional mnemonic key for the "Exit" menu bar item is "x"; when the F10 key is pressed followed by the "x" key, a WM_COMMAND message with value MI_EXIT is generated.

Mnemonics are indicated to the user by the appropriate character within the item text being underlined. This is achieved by placing a tilde character (~) within the item text, immediately prior to the required character; for example:

```
MENUITEM "E~xit",     MI_EXIT, MIS_TEXT
```

When the resource script file is compiled using the resource compiler, the menu bar item is created with the appropriate mnemonic.

9.1.3.2 Accelerator Keys

Accelerator keys or key sequences may be used to represent a pulldown menu item and provide a "fast path" to a particular command. Note that accelerator keys are not used to represent menu bar entries, since the use of an accelerator key sequence is typically more complex than the use of a mouse or an F10 + single character operation. The definition of accelerator keys is described in 9.1.5, "Accelerator Tables" on page 196. It is conventional to display an accelerator key sequence, along with the command represented by that sequence, in the pulldown menu, thus providing the user with a visual indication of the accelerator key sequence. This may be achieved by the use of the "\t" or "\a" control codes within the item text. The "\t" code causes text to the right of the code to be left-justified in a new column, whereas the "\a" code causes text to the right of the code to be right-justified in a new column.

To display an accelerator key sequence in a pulldown menu, it is conventional to use the "\t" control code. For example:

```
MENUITEM "~Tile\tShift+F5",     MI_TILE, MIS_TEXT
```

This would result in the item text "Tile" (with the "T" underscored to represent the mnemonic) being displayed in the left of the pulldown menu with the text "Shift+F5" being left-justified in a second column to the right of the item text.

9.1.4 String Tables

Tables of text strings may be defined within a resource script file for use by an application. A string table is defined using the STRINGTABLE keyword, as shown in Figure 87.

```
STRINGTABLE MAIN     PRELOAD
BEGIN
     STR_MAINTITLE,     "Application Main Window"
     STR_LIST1,         "List of Objects"
     STR_MSGTITLE1,     "Title for Message Box"
END
```

Figure 87. String Table Resource Definition

String tables may be used to contain titles, messages and other common text used by an application. The external definition of these strings makes it easy to change a title or message without modifying source code. String tables may also be used to contain menu bar or pulldown menu text for dynamic insertion by an application. Special characters such as mnemonic indicators and tab characters for columnating display may be incorporated into the string definition.

The symbolic name following the STRINGTABLE keyword identifies the string table and is used as a parameter when loading strings from the resource into application buffers using the **WinLoadString()** function. The PRELOAD keyword specifies that the resource will be incorporated into the application's main .EXE file, and is to be loaded into memory immediately rather than being loaded when called by the application.

Multiple string tables may be defined by an application. Each string table must have its own symbolic name (note that the same name may be used for a string table and another type of resource such as a menu bar) and is enclosed within the BEGIN and END keywords of a STRINGTABLE statement. Each string has its own symbolic name within the string table.

As mentioned above, strings are loaded from the string table into application buffers using the **WinLoadString()** function. For example, to load the string STR_MAINTITLE from the string table MAIN into a buffer named szTitle, the function shown in Figure 88 is used.

```
ulLength = WinLoadString(hAB,              /* Load string            */
                         NULL,             /* From appl resource file */
                         STR_MAINTITLE,    /* String id in resource  */
                         sizeof(szTitle),  /* Number of characters   */
                         szTitle);         /* Target buffer          */
```

Figure 88. Loading a Text String Resource

The **WinLoadString()** function returns an unsigned integer representing the number of characters loaded into the target buffer. Once loaded, the buffer may then be manipulated using standard programming language functions, or used as a parameter to other Presentation Manager function calls.

9.1.5 Accelerator Tables

Accelerator keys are single keys or key sequences that are used to represent a particular command (typically a pulldown menu item) within an application, and provide a fast path for the entry of that command. Accelerators are defined for an individual window and are active whenever that window is active. According to Systems Application Architecture CUA conventions, accelerator keys should be indicated to the user by placing the accelerator key sequence alongside the command in the pulldown menu. Accelerator keys are defined in the resource script file using the ACCELTABLE keyword, as shown in Figure 89.

```
ACCELTABLE CHILD1
BEGIN
     VK_F3,     MI_EXIT, VIRTUALKEY
     VK_F5,     MI_TILE, VIRTUALKEY, SHIFT
     "D",       MI_DELETE, CHAR, CONTROL
END
```

Figure 89. Accelerator Table Resource Definition

The symbolic name following the ACCELTABLE statement identifies the accelerator resource, and is passed as a parameter to the **WinCreateWindow()** or **WinCreateStdWindow()** functions when the FCF_ACCELTABLE style attribute is specified for the frame window.

In the above example, the F3 key is defined as a virtual key that when pressed will generate a WM_COMMAND message with the value MI_EXIT. This is equivalent to the user having selected the "Exit" option from the menu bar. The Shift+F5 key sequence is also defined as a virtual key that will generate a WM_COMMAND message with the value MI_TILE. Note that the shifted state of the key is indicated by use of the SHIFT option. The Ctrl+D key sequence has also been defined to generate a WM_COMMAND message with the value MI_DELETE. The Ctrl state of the key is indicated by the use of the CONTROL option (in a similar manner to the SHIFT option on the previous line). The various options for defining accelerator keys are documented in the *IBM OS/2 Version 2.0 Presentation Manager Reference.*

As already mentioned, an accelerator table is associated with a particular window by specifying the resource identifier as a parameter to the **WinCreateWindow()** or **WinCreateStdWindow()** functions. In addition, the **WinLoadAccelTable()** function may be used to dynamically load an accelerator table into memory. The **WinLoadAccelTable()** function returns the handle of the accelerator table in memory, which may then be passed as a parameter to the **WinSetAccelTable()** function to activate the accelerator table for a particular queue or window.

9.1.6 Help Tables

Help tables are used by the IPF to relate each display window, dialog box or control window to the help panel containing information about that window. Help tables and their definition are described in detail in Chapter 15, "Adding Online Help and Documentation."

9.1.7 Window and Dialog Templates

Templates defining standard windows and dialog boxes may be defined within the resource script file. Typically, a window or dialog template is designed using the Dialog Box Editor application supplied with the *IBM Developer's Toolkit for OS/2 2.0*, and is saved in a text file with an extension .DLG which is included in the resource script file with an *rcinclude* statement. A window or dialog template may also be defined directly into the resource script file. In either case, the template is defined using the WINDOWTEMPLATE or DLGTEMPLATE keywords. These keywords are actually synonymous, and the resource compiler interprets either keyword in the same way.

Within a single window or dialog template, there may be multiple WINDOW or DIALOG statements that define individual windows or dialog boxes. The nesting of the statements defines the parent/child window hierarchy. Figure 90 shows an example of nested windows within a window template.

```
WINDOWTEMPLATE WCP_0001
BEGIN
     FRAME "Window Class X", 1, 10, 10, 320, 130
     CTLDATA FCF_STANDARD
     BEGIN
          WINDOW "", FID_CLIENT, 0,0,0,0, "MyClass", 01
     END
END
```

Figure 90. Window Template Resource Definition

The window template WCP_001 contains a frame window with the title "Window Class X" and with size and positional coordinates as specified. The style attributes of the frame window are specified using the CTLDATA statement. The client window for this frame window is created using the WINDOW keyword nested within the window template, with no window title, the identifier FID_CLIENT, no size or positional coordinates (these are defined by the frame window), the class "MyClass" and the default client style.

The use of the WINDOWTEMPLATE keyword and WINDOW statements is a useful way for an application developer to predefine particular window types and styles for use by one or more applications. The template definitions may be used to create modal dialog boxes, which are loaded into memory and executed by the use of **WinLoadDlg()** and **WinProcessDlg()** calls. Definitions may also be created for standard windows or modeless dialog boxes, which are loaded into memory using the **WinLoadDlg()** function and executed by making the window or dialog box visible using the **WinShowWindow()** function.

Predefinition of windows is particularly useful when applied to dialog boxes. Here, the number and complexity of control window definitions is often such that creating such windows dynamically is a complicated task. A dialog box is defined in the resource script file (or a .DLG file, which is incorporated into the resource script file using the *rcinclude* statement) using the DLGTEMPLATE keyword.

Within a dialog template, there may be multiple dialogs defined using the DIALOG statement, and each dialog box may have multiple control windows defined using CONTROL keywords. Figure 91 on page 198 shows an example of a dialog template containing a dialog box with several control windows:

```
DLGTEMPLATE DC_CREATE
BEGIN
     DIALOG "Create an Object", DC_CREATE, 22, 32, 260, 76,,
     FCF_TITLEBAR | FCF_DLGBORDER
     BEGIN
          CONTROL "Enter the Object Name", -1, 7, 59, 246, 8, WC_STATIC,
          SS_TEXT | DT_CENTER | DT_TOP | WS_GROUP | WS_VISIBLE
          CONTROL "", 91, 43, 149, 8, WC_ENTRYFIELD,
          ES_MARGIN | ES_LEFT | WS_TABSTOP | WS_VISIBLE
          CONTROL "Enter", DID_OK, 38, 5, 38, 12, WC_BUTTON,
          BS_PUSHBUTTON | BS_DEFAULT | WC_TABSTOP | WC_VISIBLE
          CONTROL "Cancel", DID_CANCEL, 38, 5, 38, 12, WC_BUTTON,
          BS_PUSHBUTTON | WC_TABSTOP | WC_VISIBLE
     END
END
```

Figure 91. Dialog Template Resource Definition

The dialog template is equivalent to a frame window, and is named DC_CREATE. This symbolic name is used to identify the dialog resource and is passed as a parameter to the **WinDlgBox()** function, which loads and processes the dialog box.

The dialog box is defined with a title bar and a dialog border, and is also named using the symbolic name DC_CREATE. The dialog box contains a static text control window providing instructions to the user, and an entry field into which the user may enter text. It also contains an "Enter" and a "Cancel" pushbutton.

Note that the resource identifier for the static text string does not use a symbolic constant, but simply has the value "-1." This is done because there is no need for the application to access the text string; it is merely present as a prompt to the user. It is therefore conventional to omit the symbolic constant and use "-1" as the value. Multiple text strings may have the same value.

9.2 Resource Script File

The resource script file is an ASCII text file in which Presentation Manager resources are either defined or referenced. A sample resource script file is given in Figure 92 on page 199.

Note that the dialog templates are not defined directly in the resource script file, but are incorporated at the end of the resource script file using an *rcinclude* statement for the file *mydlg.dlg*. This is the typical way to incorporate dialog templates that are created by the Dialog Box Editor and stored in a DLG file.

```
#include <os2.h>
#include "myappl.h"
#include "mydlg.h"

ICON     MAIN     APPLIC.ICO
ICON     CHILD1   CHILD1.ICO

BITMAP   INTRO    LOGO.BMP

STRINGTABLE MAIN     PRELOAD
BEGIN
     STR_MAINTITLE,     "Application Main Window"
     STR_LIST1,         "List of Objects"
     STR_MSGTITLE1,     "Title for Message Box"
END

MENU MAIN      PRELOAD
BEGIN
     SUBMENU "~File",  MI_FILE, MIS_TEXT
     BEGIN
          MENUITEM "~New...",  MI_NEW, MIS_TEXT
          MENUITEM "~Open",    MI_OPEN, MIS_TEXT
          MENUITEM "~Save",    MI_SAVE, MIS_TEXT
          MENUITEM "Save ~as", MI_SAVEAS, MIS_TEXT
     END
     SUBMENU "~Edit", MI_EDIT, MIS_TEXT
     BEGIN
          MENUITEM "Cu~t",     MI_CUT, MIS_TEXT
          MENUITEM "~Copy",    MI_COPY, MIS_TEXT
          MENUITEM "~Paste",   MI_PASTE, MIS_TEXT
     END
     MENUITEM "E~xit",         MI_EXIT, MIS_TEXT
     MENUITEM "~Help",         MI_HELP, MIS_HELP |
                                       MIS_BUTTONSEPARATOR
END

ACCELTABLE MAIN
BEGIN
     VK_F3,     MI_EXIT, VIRTUALKEY
     VK_F5,     MI_TILE, VIRTUALKEY, SHIFT
     "D",       MI_DELETE, CHAR, CONTROL
END
ACCELTABLE CHILD1
BEGIN
     VK_F1,    MI_HELP, VIRTUALKEY, HELP
     VK_F3,    MI_EXIT, VIRTUALKEY
     "D",      MI_DELETE, CHAR, CONTROL
END

rcinclude MYDLG.DLG
```

Figure 92. Resource Script File

The resource script file has a number of *#include* statements at the start, similar to those typically found in C programs. This is because the symbolic names used throughout the resource script file represent integer constants, and must be defined in the application's header file *myappl.h*. Other symbolic names may be used in the .DLG file, and must also be defined; the header file *mydlg.h* for

these symbolic names is generated by the Dialog Box Editor. Finally, a number of symbolic names such as DID_OK and DID_CANCEL are actually defined by Presentation Manager rather than by the application, and therefore the file *os2.h* is also required.

The resource script file is used as input to the resource compiler provided as part of the *IBM Developer's Toolkit for OS/2 2.0*. For further information on the resource compiler and its operation, see 14.4, "Resource Compilation" on page 280.

9.3 Using Resources

As mentioned throughout this chapter, resources are typically loaded and used in an application by specifying the symbolic name of the resource as a parameter to a function that requires the resource. The resource is then loaded and used by that function in performing its task. However, there are several ways in which the resource may be loaded, depending upon where it resides. These are discussed in the following sections.

9.3.1 Loading From Within the Application

In the typical case, resources are incorporated into an application by passing the resource script file to the resource compiler. The resource compiler compiles the resource definitions and incorporates them into an executable file that has already been created.

Many of the functions that require a resource identifier, such as **WinLoadString()** and **WinLoadPointer()**, also accept the identifier of a resource file as one of their parameters. For resources that are incorporated into the application's .EXE file, this parameter should be specified as NULL. For example, to load a pointer from a resource defined within the .EXE file, the following call is used:

```
hPointer = WinLoadPointer(hDesktop,     /* Desktop handle          */
                          NULL,         /* Within .EXE file        */
                          DRAW);        /* Resource symbolic name  */
```

Other Presentation Manager functions that use this convention include **WinLoadAccelTable()**, **WinLoadMenu()** and **WinCreateStdWindow()**.

9.3.2 Loading Resources From a DLL

Presentation Manager resources may also be defined and stored in a dynamic link library. The process of compiling and placing resources in a DLL is described in 14.5.3, "Presentation Manager Resources in a DLL" on page 282. Once a resource is located in a DLL however, the DLL module must be loaded into memory by the application, and a module handle obtained at run time before the resource may be accessed by a Presentation Manager function. This is typically achieved using the **DosLoadModule()** or **DosGetModuleHandle()** functions. Figure 93 on page 201 illustrates the necessary code to load a dynamic link library named MYDLL from a directory identified in the LIBPATH statement in CONFIG.SYS, and to load a string resource from this DLL.

```
rc = DosLoadModule(NULL,                    /* No object name          */
                   0,                       /* No object length        */
                   "MYDLL",                 /* DLL module name         */
                   hModule);                /* DLL handle (returned)   */

ulLength = WinLoadString(hAB,               /* Load string             */
                         hModule,           /* DLL module handle       */
                         STR_TITLE,         /* Resource ID within DLL  */
                         sizeof(szTitle),   /* Number of bytes         */
                         szTitle);          /* Target buffer           */
```

Figure 93. Loading Resources From a DLL

The **DosLoadModule()** function call loads the dynamic link library with the name "MYDLL" (the default extension of .DLL is assumed) into memory and returns a handle *hModule* of type HMODULE. This handle is then passed as the resource file identifier to the **WinLoadString()** function call, which accesses the resources within the module. Other function calls such as **WinLoadPointer()** work in a similar manner.

9.3.3 Loading Dialogs From a DLL

The **WinDlgBox()** function also allows a DLL module handle to be specified, and thus enables dialog template definitions to be loaded from a DLL. For instance, to load and create a dialog box from a dialog template resource DC_001 defined in a DLL module named WINDLL.DLL, the following call sequence is used:

```
rc = DosLoadModule(NULL,                    /* No object name          */
                   0,                       /* No object length        */
                   "MYDLL",                 /* DLL module name         */
                   hModule);                /* DLL handle (returned)   */

rc = WinDlgBox(hDesktop,                    /* Desktop is parent       */
               hFrame,                      /* Frame is owner          */
               dpProc001,                   /* Dialog procedure address */
               hModule,                     /* DLL module handle       */
               DC_001,                      /* Resource ID within DLL  */
               NULL);                       /* No create parameters    */
```

Note that if the dialog procedure *dpProc001* to be associated with this dialog box is also defined within the DLL module, the address of this procedure must be obtained by the application before the **WinDlgBox()** call is issued. This is achieved using the **DosGetProcAddr()** function, which returns the address of the required function, as shown in the following example:

```
rc = DosGetProcAddr(hModule,
                    "Proc1",
                    dpProc001);
```

In this case, *Proc1* is the name of the required entry point in the DLL module, and *dpProc001* is a variable of type PFNWP which contains the address of the procedure returned by the **DosGetProcAddr()** call. While the address of the dialog procedure could have been supplied implicitly by using load-time rather than run-time dynamic linking, run-time dynamic linking is necessary to load the dialog box resource, and it is logical to place the resource and its associated dialog procedure in the same DLL module. An example of the complete procedure required to load a dialog box from a DLL is given in Figure 94 on page 202.

```
BOOL CustInfoDialog()
{
     HMODULE hModule;                        /* DLL module handle       */
     PFNWP   dpDlgProc;                      /* Dialog procedure addr   */
     USHORT  usResult;                       /* Result storage          */

     DosGetModuleHandle("WINDLL",            /* Get DLL module handle   */
                        hModule);

     DosGetProcAddr(hModule,                 /* Get address of dialog   */
                    "dpCustDlg",             /* procedure               */
                    dpDlgProc);

     rc = WinDlgBox(HWND_DESKTOP,            /* Load & process dialog   */
                    NULL,                    /* No owner                */
                    dpDlgProc,               /* Dialog procedure addr   */
                    hModule,                 /* DLL module handle       */
                    DC_CUSTDLG,              /* Dialog template id      */
                    NULL);                   /* No create parameters    */

     return(usResult);
}
```

Figure 94. Loading a Dialog Resource From a DLL

When loading dialogs from DLL modules, it is recommended that a combination of load-time and run-time dynamic linking techniques be used. A calling routine should be placed in the DLL which, in response to an application request, loads and obtains the appropriate module handle, obtains the required dialog procedure address and executes the dialog. This relieves the application of the responsibility for loading the dynamically-linked resources and routines. An example of such a routine is given in Figure 94. The calling routine *CustInfoDlg* is defined as an entry point within the DLL module, since it will be called from the application's main executable module. An import library is then built for the DLL, and linked with the application code using standard conventions for load-time dynamic linking.

When *CustInfoDlg* is invoked by the application, it obtains a module handle for its own DLL module, which has already been loaded when the call to *CustInfoDlg* was made, and uses this handle to obtain the address of the required dialog procedure using standard run-time dynamic linking conventions. It then issues a **WinDlgBox()** call to load and process the dialog box, and returns the result to the application. This example illustrates the combination of load-time and run-time dynamic linking conventions.

9.4 Resources and National Language Support

Since Presentation Manager resources provide the ability to define all the user interface properties of a Presentation Manager application, externally to the application code, they provide a useful means for implementing national language support within Presentation Manager applications. Resources may be used to define:

- Window titles
- Menu bar and pulldown menu entries, including mnemonics and accelerator keys

- Dialogs
- Messages
- Symbols such as icons and pointers.

In short, all of the language-specific properties of an application may be defined using Presentation Manager resources. Icons and other graphical symbols used by the user interface may also be tailored to suit different cultures where such symbols may have different meanings.

The set of resources for each national language may be compiled and incorporated into a separate dynamic link library, which may be accessed by the application in order to load the required resources, as described in 9.3.3, "Loading Dialogs From a DLL" on page 201. The resource identifiers must, of course, be identical in each DLL. Upon installation of the application on a workstation, an installation procedure can prompt the user to determine the required language, and install the appropriate DLL for that language.

Where multiple languages must be supported in the same system, an application may query the codepage currently in use by its parent process using the **WinQueryCp()** function, and load resources from a specific DLL, depending upon the result of the function call. While this method is by no means foolproof, it will suffice for many languages that use a single national codepage and single-byte characters.

9.5 Summary

It can be seen that Presentation Manager provides the mechanism by which an application developer may externally define the user interface properties of his/her application. This ability provides the benefit that these external properties may be modified, or different versions substituted, without the need to modify the application code itself. In addition, standard user interface objects such as icons, pointers and dialog boxes, along with their associated dialog routines, may be defined and stored in dynamic link libraries for use by multiple applications.

Resources are defined using a resource script file, which is an ASCII text file containing definitions for text-based resources and references to other files that contain definitions for non-textual resources such as pointers and icons. The resource script file is used as input to the resource compiler, which compiles resource definitions and incorporates them into an executable module.

Resources may be incorporated into the application's main .EXE file, or may be stored in a dynamic link library and loaded into memory using run-time dynamic linking. Application procedures such as dialog procedures, which are associated with such resources, may also be defined and stored in the same DLL module, thus providing the opportunity to create libraries of standard resources, including standard dialogs, which may be used by multiple applications.

Chapter 10. Multitasking Considerations

Systems Application Architecture CUA guidelines recommend that an application should complete the processing of a user- or system-initiated event within 0.1 seconds and be ready to continue interaction with the end user. The particular implementation of the message handling concept under Presentation Manager means that the application's primary thread must complete the processing of a Presentation Manager message *before* any further messages can be passed to applications; thus it is possible for the user to be "locked out" of the system if an application does not complete its processing within a reasonable period of time.

While the 0.1 second time period is adequate for the processing of most events, it may be insufficient for those that result in lengthy processing such as access to a remote system. It is therefore recommended that any window procedure performing some processing that is likely to take longer than 0.1 seconds to complete should carry out this processing using a separate thread of execution under OS/2. The application's primary thread may then initiate the secondary thread and immediately return control to Presentation Manager, thereby enabling the primary thread to continue with user interaction.

The separation of processing into a primary and one or more secondary threads may occur in a number of ways:

- Where the window procedure is an object window procedure, and the majority of its methods may result in lengthy processing, the window procedure itself may be implemented in a secondary thread.
- Where only a single method results in lengthy processing, or where the window procedure is concerned with a display object, a single subroutine containing that method may be started in a secondary thread.

In certain circumstances where the different portions of an application's task are entirely self-contained, and where it is desirable to isolate the portions from one another, the application may be divided into separate processes. Division of the application in this way means that each portion resides and executes in its own address space, fully protected from other portions of the application. This approach is particularly useful for applications that exploit the Workplace Shell, since the implementation of the Workplace Shell in OS/2 Version 2.0 causes Workplace Shell objects to execute, by default, under the control of the Workplace Shell process. The use of multiple processes within an application provides better protection for resources used by Workplace Shell objects.

Note that for performance reasons, the use of multiple threads within the same process is preferable to the use of multiple processes. This is because switching between threads involves far less system overhead than switching between processes.

Processes and threads may communicate with one another in a number of ways for the purposes of exchanging information, and for synchronizing execution flow and access to data objects. The techniques of communication between threads and processes are described in 10.5, "Communicating With a Secondary Thread" on page 215 and 10.6, "Communicating With Another Process" on page 216.

Maintaining synchronization between threads and processes is discussed in 10.7, "Maintaining Synchronization" on page 229.

10.1 Creating a Secondary Thread

In 4.3, "Application Structure" on page 43, it is mentioned that an application must create its own input message queue to process messages intended for its windows. The Presentation Manager message-handling implementation creates message queues on a per-thread basis, and thus requires that any thread that creates a window (whether that window is a display window or an object window) and processes messages must have its own message queue.

The primary thread of an application is typically a user interface thread that handles processing for display windows on the screen; this thread creates the application's main message queue and processes messages caused by user interaction. The primary thread may also create a secondary thread to deal with messages that cause lengthy processing to be carried out, leaving the primary thread free to respond to user input. A secondary thread may be created in one of two ways:

- The **_beginthread()** function provided by the C compiler should be used to create secondary threads that will contain object windows, or that contain code which calls C run-time library functions. This function maintains certain internal C library control data that is, by default, not maintained by the **DosCreateThread()** function.
- The **DosCreateThread()** function provided by OS/2 may be used to create secondary threads that will not contain object windows, and that do not call C run-time library functions. The **DosCreateThread()** function offers a slight performance advantage over the **_beginthread()** function.

The **_beginthread()** function is used since it establishes internal semaphores to serialize access to the run-time library's global data and non-reentrant library functions, transparently to the calling application. The **_beginthread()** function also maintains information for each thread, such as the exception handling environment and the calling address for reentrant functions. Since window procedures are reentrant, use of **DosCreateThread()** in such situations may cause execution errors.

Whenever a thread is created, a **thread information block** (**TIB**) is created by the operating system. The TIB contains information such as the thread ID, priority and stack size. This information may be accessed by the application using the **DosGetInfoBlocks()** function. This function also returns a pointer to information on the thread's parent process, which resides in the **process information block** (**PIB**). The **DosGetInfoBlocks()** function is described in the *IBM OS/2 Version 2.0 Control Program Reference*.

10.1.1 Threads Containing Object Windows

In the case where the processing of an event involves access to and manipulation of another data object, the secondary thread should create its own message queue and one or more object windows with window procedures to process any messages passed by the primary thread. This technique preserves the object-oriented nature of the application by isolating data objects from one another.

The primary thread then passes messages to the secondary thread's object windows, in an identical manner to that used when passing messages to a window procedure in the primary thread; it is recommended that this be achieved using the **WinPostMsg()** function, since this call allows asynchronous processing of the message and preserves the correct serialization of messages in the system, as described in 4.3.2.1, "Invoking a Window Procedure" on page 47.

Asynchronous threads with object windows are normally created by window procedures in the application's primary thread issuing a **_beginthread()** function call. This call is typically made by a window procedure during its processing of the WM_CREATE message; the secondary thread and its object window (or windows) are then initialized and able to accept any requests passed to them. An example of the **_beginthread()** function is shown in Figure 95.

```
#define  STACKSIZE    8192
  :
  :
case WMP_DOLONGTASK:
     _beginthread(thread,            /* Entry point of thread routine   */
                  &Stack,            /* Pointer to stack memory object  */
                  STACKSIZE,         /* Size of stack memory object     */
                  (PVOID)hwnd);      /* Initialization data for thread  */
     break;
```

Figure 95. Creating a Thread With an Object Window

Note that when using the IBM C Set/2 compiler, the second parameter in the **_beginthread()** function call (the pointer to the stack) is ignored, since the **_beginthread()** function automatically allocates memory for the stack. This parameter is included merely to allow source code compatibility with applications written for the earlier IBM C/2 and Microsoft C compilers, which required the application to explicitly allocate stack space for a secondary thread. Note that the minimum recommended stack size for a thread containing object windows is 8192 bytes (8 KB).

The handle of the window from which the secondary thread is being created is passed to the thread in the **_beginthread()** call. The secondary thread's main routine may then pass this handle to the object window created in the thread. Upon successful creation, the object window can then pass an acknowledgement message back to the window that created it, containing the handle of the object window, in order that the calling window may then post messages to the object window. This acknowledgement message also ensures that the object window is correctly created and initialized before any messages are posted to it.

The secondary thread's main routine is similar in structure to the main routine of the primary thread in a Presentation Manager application. The main routine registers the object window class, creates a window of that class and enters a message processing loop.

It should be noted that the secondary thread's main routine is identical to that of the application's primary thread, with the exception that a secondary thread need not register itself to Presentation Manager, since this is typically done once per application, by the primary thread. However, if certain functions such

as error processing are required on a per-thread basis, a separate anchor block must be created for the secondary thread, and hence an additional **WinInitialize()** call must be made.

Figure 96 shows a secondary thread that registers an object window class and creates a window of that class.

```
void thread(HWND hWnd)
{
    HMQ  hMsgQ;                          /* Message queue handle      */
    HWND hWindow;                        /* Window handles            */
    QMSG qMsg;                           /* Message queue structure   */

    WinRegisterClass(hAB,                /* Register window class     */
                     WCP_OBJECT,         /* Class name                */
                     (PFNWP)wpObject,    /* Window procedure          */
                     0L,                 /* No class style            */
                     0);                 /* No extra window words     */

    hObject=WinCreateWindow(HWND_OBJECT, /* Create object window      */
                            WCP_OBJECT,  /* Object handle is parent   */
                            (PSZ)0,      /* No window text            */
                            0L,          /* No style                  */
                            0,0,0,0,     /* No position               */
                            NULL,        /* No owner                  */
                            HWND_TOP,    /* On top of siblings        */
                            0,           /* No window id              */
                            hWnd,        /* Handle in WM_CREATE       */
                            0);          /* No pres. parameters       */

    while (WinGetMsg(hAB,                /* Loop until WM_QUIT        */
                     &qMsg,
                     (ULONG)0,
                     0, 0))
          WinDispatchMsg(hAB, &qMsg);

    WinDestroyWindow(hObject);           /* Destroy object window     */
    WinDestroyMsgQueue(hMsgQ);           /* Destroy message queue     */
    _endthread();                        /* Terminate thread          */
}
```

Figure 96. Secondary Thread Creating an Object Window

An object window is created using the normal **WinCreateWindow()** call, as illustrated in Figure 96. The window's parent is specified as the conceptual object window, the handle of which is obtained from the **WinQueryObjectWindow()** function, or using the defined constant HWND_OBJECT. Note that the handle of the window that created the thread is passed to the object window in the *CtlData* parameter of the **WinCreateWindow()** call, in order that the object window may pass its handle back to the calling window to indicate its readiness to receive messages.

The secondary thread retrieves messages from its input queue in the conventional manner using **WinGetMsg()**, and invokes Presentation Manager using **WinDispatchMsg()** to pass the message to its object window procedure. Thus a secondary thread has a message processing loop similar to that of the application's (primary thread's) main routine.

An object window procedure is identical in structure to a "normal" display window procedure. An example of an object window procedure is illustrated in Figure 97 on page 209.

```
MRESULT EXPENTRY wpObject(HWND hWnd,
                          ULONG ulMsg,
                          MPARAM mp1,
                          MPARAM mp2)
{
        HWND     hNotify;
        HWND     hObject;

        switch (ulMsg)
               {
               case WM_CREATE:
                    WinDefWindowProc(hWnd,
                                     usMsg,
                                     mp1,
                                     mp2);
                    <initialize instance data>
                    <open data objects>

                    hNotify=HWNDFROMMP(mp1);
                    hObject=MPFROMHWND(hWnd);
                    WinPostMsg(hNotify,
                               WMP_NOTIFY,
                               hObject,
                               0);

                    return((MRESULT)FALSE);
                    break;
               case WMP_PUTDATA:
                    <put data into database>
                    <post message to logging object>
                    return((MRESULT)TRUE);
                    break;
               case WMP_GETDATA:
                    <get data from database>
                    <post data to caller in message>
                    return((MRESULT)TRUE);
                    break;
               case WM_DESTROY:
                    <close data objects>
                    <free any instance data areas>
                    return((MRESULT)0);
                    break;
               default:
                    return(WinDefWindowProc(hWnd,
                                            ulMsg,
                                            mp1,
                                            mp2));
               }
}
```

Figure 97. Sample Object Window Procedure

Upon creation, an object window receives a WM_CREATE message in the same way as a standard window. The window may capture and explicitly process this

message in order to open or create data objects, initialize instance data, etc., as illustrated in Figure 97. Once opened, however, an object window typically only receives a number of application-defined messages requesting certain actions on data objects owned by the window.

The object window procedure shown in Figure 97 also extracts the handle of the window that issued the **WinCreateWindow()** function call, which is normally passed to the object window as part of the WM_CREATE message. The window procedure then uses this handle to send an acknowledgement message back to this window, containing its own window handle and thus enabling the two windows to communicate with one another. This is necessary when object windows are created in secondary threads, as described in 10.5, "Communicating With a Secondary Thread" on page 215. Note that the object window procedure must use the *system* linkage convention; this is achieved using the EXPENTRY keyword.

The only other system-defined message class normally received by an object window is the WM_DESTROY message class passed to the window by Presentation Manager when a **WinDestroyWindow()** call is issued by the thread. An object window should respond to the WM_DESTROY message by closing, destroying or freeing any data objects to which it has obtained access, and backing out any uncommitted units of work.

Figure 97 on page 209 shows a number of application-defined message classes being processed by the object window procedure. These message classes are typically defined by the application during its initialization. For message classes that will be processed by an object window procedure loaded from a DLL module, the message classes should be defined in the include file for that DLL module, rather than explicitly within the application that uses the DLL module. This further enhances the isolation of the internal workings of the object window from other components of the application, and facilitates reusability of the code.

The window procedure in the primary thread must have some way of determining when the object window has completed its processing, at which point it may safely assume that the previous event has been processed successfully and allow the user to continue operating upon the data object, or take appropriate error action. This indication may be provided in a number of ways, which are discussed in 10.7, "Maintaining Synchronization" on page 229.

10.1.2 Threads Without Object Windows

When the processing to be performed within a secondary thread is limited in scope to the data objects "owned" by the current application object, an object window is not warranted. A secondary thread without an object window is similar in both appearance and behavior to a normal subroutine. However, the routine executing in the secondary thread performs its tasks asynchronously to the primary thread, although it still has access to the same data objects. Such a thread is typically started when required by issuing a **DosCreateThread()** call from within a window procedure in the primary thread. A sample invocation of such a thread is illustrated in Figure 98 on page 211.

```
case WMP_THREAD:
     usReturn=DosCreateThread(ThreadID,     /* Create thread              */
                ThreadRoutine,              /* Routine to run in thread   */
                &InitData,                  /* Initialization data        */
                0L,                         /* Start immediately          */
                4096);                      /* Stack size for thread      */
     break;
```

Figure 98. Creating a Thread Without an Object Window

Two considerations arise when processing asynchronous threads without the use of object windows:

- The primary thread must not attempt to access a data object at the same time as a secondary thread is updating that data object, since the state of the data during the update is undetermined and unpredictable results could occur.
- The primary thread must have some way of determining when the secondary thread has completed its processing, at which point it may then access the data objects that were manipulated by the secondary thread.

These two conditions may be achieved by adopting a convention whereby a secondary thread has exclusive access to its data object(s) for the duration of its execution. It is therefore only necessary for the primary thread to determine when the secondary thread has completed processing, at which point it may access the data objects.

Where this is not possible, mutex semaphores may be used to serialize access to resources such as data objects. Each thread that requires access must bid for the semaphore. If the semaphore is already held by another thread, the requesting thread must wait for that thread to release the semaphore before attempting to access the resource.

A number of mechanisms for synchronizing execution and/or access to resources are described in 10.7, "Maintaining Synchronization" on page 229.

10.2 Creating Another Process

Each application running under OS/2 typically resides in its own process, and therefore has its own address space. Resources created by or allocated to a process are normally private to that process. If required for application purposes, this process may in turn create one or more additional processes to perform part of the application's processing. Additional processes may be created in either of two ways:

- As a child process of the creating process, in which case the child process will automatically terminate upon termination of the creating process. Such processes are started using the **DosExecPgm()** function.
- As a separate process, in which case the process will not automatically terminate when its creator terminates, and must be explicitly terminated either by its creator or by another process in the system. Such a process is started using the **DosStartSession()** function.

When an application uses multiple processes, it is usual for the first process to be regarded as the "primary" process for the application, and for other

processes to be started as children of this process. This is conceptually similar to the use of primary and secondary threads.

It is therefore conventional to use the **DosExecPgm()** function to start a child process. This function is illustrated in Figure 99.

```
CHAR        szClient[7];                /* ASCII form of win handle   */
CHAR        LoadError[100];             /* Buffer for failure reason  */
RESULTCODES ReturnInfo;                 /* Returned info from call    */
PID         pidServer;                  /* Child process id           */
APIRET      rc;                         /* Return code                */
  :
  :
itoa(hWnd, szClient);                   /* Convert handle to ASCII    */

rc = DosExecPgm(LoadError,              /* Start child process        */
                sizeof(LoadError),      /* Size of buffer             */
                EXEC_ASYNCRESULT,       /* Execute asynchronously     */
                szClient,               /* Window handle in ASCII     */
                0,                      /* No new environment vars    */
                &ReturnInfo,            /* Returned info address      */
                "server.exe");          /* Name of program to start   */

pidServer = ReturnCodes.termcodepid;

<Store pidServer in window words>
```

Figure 99. Starting a Child Process. This example shows the use of the **DosExecPgm()** *function to start a process from another process within the application.*

The window handle of the window from which the **DosExecPgm()** call is made, is passed to the child process as an argument, using the the fourth parameter of the **DosExecPgm()** function. This enables the child process to post a message to its parent upon completing its initialization, indicating the window handle of its own window. In this way, communication via Presentation Manager messages may be established in both directions.

Note that since the fourth parameter to the **DosExecPgm()** function is defined as an ASCII string, the window handle is converted to its ASCII equivalent before the call is issued. The main routine of the child process subsequently converts the handle back to its true form.

Use of the EXEC_ASYNCRESULT flag in the **DosExecPgm()** call causes the operating system to save the termination codes of the child process so that they may be examined at a later point by the parent process, using a **DosWaitChild()** function call for synchronization purposes (see 10.7.4, "DosWaitChild() Function" on page 234 for further information).

The process ID of the child process is returned by the **DosExecPgm()** function as part of the RESULTCODES structure in the sixth parameter. This value should be stored by the parent process, since it is used if and when the parent process needs to terminate the child at some later point during execution.

10.3 Destroying a Secondary Thread

Secondary threads should be terminated when they are no longer required, to reduce the context-switching overhead of the operating system. The method of termination depends upon how the secondary thread was created, and whether or not it has created object windows.

10.3.1 Threads Containing Object Windows

A secondary thread with an object window should be terminated by the window procedure in the primary thread that initially created the secondary thread. This is achieved simply by posting a WM_QUIT message to the object window, which will cause the message processing loop for the secondary thread to terminate.

The thread's main routine then issues **WinDestroyWindow()** calls for its object windows. These calls cause WM_DESTROY messages to be posted to the object windows, which will process these messages in order to close data objects, release any resources, etc., in accordance with established conventions.

Once all data objects and other resources have been released or destroyed, the thread should terminate itself using the **_endthread()** function. The use of this function will ensure that the semaphores and control structures used by the **_beginthread()** function are correctly reset.

10.3.2 Threads Without Object Windows

Secondary threads without object windows are typically created to perform a lengthy processing operation within the scope of a single event. A window procedure under the control of the primary thread creates the secondary thread to process a particular subroutine, and the secondary thread terminates automatically when this subroutine reaches an exit point. However, the **DosExit()** function should be called as the last action in the secondary thread, in order to ensure that the memory allocated for the thread's stack is correctly released by the operating system.

However, some checks may be necessary to ensure orderly termination of a secondary thread, particularly where access to data objects is involved. Upon termination of an application's primary thread, all secondary threads that have not already been terminated by the application are forcibly terminated by OS/2. Where the secondary thread's processing involves a critical data operation such as the update of a database, the primary thread should ensure that the secondary thread has completed its processing before allowing itself to terminate.

It is recommended that before creating a secondary thread without an object window, a window procedure should set an event semaphore, and pass the handle of this semaphore to the secondary thread. The event semaphore is then cleared by the secondary thread as the *last action* before it terminates.

Upon receiving a WM_DESTROY message, the window procedure in the primary thread should test the state of the event semaphore and wait for the semaphore to clear before completing the WM_DESTROY message processing (which should include releasing the semaphore) and returning control to Presentation Manager. This will ensure that the secondary thread terminates in an orderly manner before the primary thread is terminated.

The use of event semaphores is described in more detail in 10.7, "Maintaining Synchronization" on page 229.

10.3.3 Forcing Termination of a Thread

In certain circumstances, it may be necessary to terminate a secondary thread without waiting for the thread to complete its processing; for example, the user may decide to exit from the application. This capability is provided under OS/2 Version 2.0 using the **DosKillThread()** function. This function is illustrated in Figure 100.

```
<Retreive thread id from window words>

usReturn = DosKillThread(ThreadID);   /* Destroy secondary thread        */
```

Figure 100. DosKillThread() Function. This function allows the forced termination of a thread.

Note that the **DosKillThread()** function cannot be used to terminate the current thread; if the application attempts to issue a **DosKillThread()** function call for the current thread, the function will return an error. To terminate a secondary thread from within that thread, the **_endthread()** function should be used if the thread was created with the **_beginthread()** function, or the **DosExit()** function may be used if the thread was created using the **DosCreateThread()** function.

10.4 Terminating a Process

A process may terminate another process running in the system, provided it has access to the process ID of the process it wishes to terminate. This process ID is returned by the **DosExecPgm()** function when the process is created and since, in the majority of cases, a process is terminated by the process that created it, this presents no particular problem since the process ID can be stored as a global variable or as instance data in window words, until it is needed to terminate the process.

A process is terminated using the **DosKillProcess()** function. This function may be used to terminate a single process, or to terminate a process and all its descendants (that is, its children, along with their children, and so on). An example of the **DosKillProcess()** function is given in Figure 101.

```
<Retrieve process id from window words>

rc = DosKillProcess(1,                    /* Kill only this process     */
                    pidServer);           /* Process ID to be killed    */
```

Figure 101. Terminating a Process. This example shows the use of the DosKillProcess() function to terminate a single process.

The value of "1" specified for the first parameter in the **DosKillProcess()** call causes the function to terminate only the specified process and not its descendants (if any).

10.5 Communicating With a Secondary Thread

The primary thread may wish to communicate with the secondary thread in order to initiate an event or transfer data. The methods available for such communication differ, depending upon whether the secondary thread contains an object window.

10.5.1 Threads Containing Object Windows

When a secondary thread is created, and in turn creates an object window as shown in Figure 96 on page 208, the handle of the calling window may be passed to the object window as part of the WM_CREATE message, and the object window procedure uses this handle to pass its own window handle back to the calling window as part of an acknowledgement message. This technique is illustrated in the sample object window procedure shown in Figure 97 on page 209.

Once the calling window receives this message and extracts the object window's handle, it should store the handle in its own instance data. It may then use the handle at any time to pass a Presentation Manager message to the object window, in order to initiate an event in the object window.

By convention, messages passed to an object window should contain the window handle of the calling window, within the message parameters. The object window procedure may then use this handle to pass an acknowledgement message or return data to the calling window in the primary thread. This technique allows the same object window to process messages from multiple sources. See 10.9, "Client-Server Applications" on page 236 for additional considerations.

Note that messages passed to object windows in secondary threads should be *posted* using the **WinPostMsg()** function, rather than being sent using the **WinSendMsg()** function. This causes asynchronous processing of the message, and allows the primary thread to return to Presentation Manager and continue interaction with the end user.

In situations where the user must be prevented from carrying out certain actions while the object window processes an event, the calling window procedure should disable those actions in the action bar immediately *before* posting the message to the object window, and re-enable those actions *only* after a completion message has been received from the object window. This prevents the user from carrying out such actions, but does not prohibit other actions within the application, or interaction with other applications on the desktop.

10.5.2 Threads Without Object Windows

Where a secondary thread is created only to process a specific event, and where the thread terminates upon completion of that event, communication between the primary and secondary threads is usually not required. Necessary data is communicated to the secondary thread as part of the **DosCreateThread()** function, including pointers to the data objects upon which the thread must operate. The secondary thread then proceeds to process the event, independently of the primary thread.

The only communication from the secondary thread to the primary thread occurs upon completion of the event, when the secondary thread signals this

completion to the primary thread. Completion may be signalled by Presentation Manager messages or via semaphores; see 10.7, "Maintaining Synchronization" on page 229 for further discussion.

10.6 Communicating With Another Process

Communication between processes must use one of the architected methods provided by OS/2, since the operating system by default prohibits different processes from accessing the same resources. For this reason, communication between processes is slightly more complex than communication between threads, but requires less care on the part of the programmer to ensure synchronization and data integrity.

The mechanisms provided by OS/2 for interprocess communication are:

- Presentation Manager messages
- Shared memory
- Queues
- Pipes (both named and anonymous)
- Atoms
- Dynamic data exchange (DDE).

Each of these mechanisms is explained in detail in the *IBM OS/2 Version 2.0 Application Design Guide*, and simple examples are given in the following sections.

10.6.1 Presentation Manager Messages

When a child process creates its own windows, Presentation Manager messages can be used to signal events and/or pass information between the parent and child processes, provided the windows' handles are known to one another. One technique for passing window handles during process creation is described in 10.2, "Creating Another Process" on page 211. Even where the child process does not create its own windows, it may use Presentation Manager messages to indicate events and pass information to its parent process.

The amount of information which can be passed in a Presentation Manager message is somewhat limited, due to the four-byte size of each message parameter. The use of Presentation Manager messages for passing information is therefore typically combined with other mechanisms such as shared memory or atoms. The message parameters are then used to carry pointers to shared memory objects, or string handles that are then used by the child process to access the required information.

10.6.2 Shared Memory

Shared memory objects may be allocated and used to pass information between specific processes. Such memory objects may be named or anonymous.

The example that follows assumes that two processes are created in the system: a client process that accepts user input and displays results, and a server process that accepts requests from the client, accesses data objects and returns the requested data to the client. Both of these processes create windows.

In order to communicate a request to the server, the client must first allocate a shared memory object, using the **DosAllocSharedMem()** function as described in

Chapter 5, "The Flat Memory Model." Since the process ID of the server process is known to the client, the client can provide access to the shared memory object for the server process, using the **DosGiveSharedMem()** function, which is also described in Chapter 5, "The Flat Memory Model." This technique is shown in Figure 102 .

```
REQUEST *Request;                              /* Request structure     */
REPLY   *Reply;                                /* Reply structure ptr   */
  :
CASE WMP_SENDREQUEST:
     rc = DosAllocShrMem(&Request,             /* Allocate memory obj   */
                         NULL,                 /* Anonymous memory obj  */
                         sizeof(REQUEST),      /* Size of memory obj    */
                         OBJ_GIVEABLE,         /* Object is giveable    */
                         PAG_WRITE |           /* Allow write access    */
                         PAG_READ  |           /* Allow read access     */
                         PAG_COMMIT);          /* Commit storage now    */
     rc = DosGiveSharedMem(Request,            /* Give access to object */
                           pidServer,          /* Process to get access */
                           PAG_WRITE |         /* Write access allowed  */
                           PAG_READ);          /* Read access allowed   */

     rc = DosAllocShrMem(&Reply,               /* Allocate memory obj   */
                         NULL,                 /* Anonymous memory obj  */
                         sizeof(REPLY),        /* Size of memory obj    */
                         OBJ_GIVEABLE,         /* Object is giveable    */
                         PAG_WRITE |           /* Allow write access    */
                         PAG_READ  |           /* Allow read access     */
                         PAG_COMMIT);          /* Commit storage now    */
     rc = DosGiveSharedMem(Reply,              /* Give access to object */
                           pidServer,          /* Process to get access */
                           PAG_WRITE |         /* Write access allowed  */
                           PAG_READ);          /* Read access allowed   */

     Request->hRequester = hWnd;               /* Set requester handle  */

     <Initialize other Request structure fields>

     WinPostMsg(hServer,                       /* Post msg to server    */
                WMP_DOREQUEST,                 /* DO_REQUEST message    */
                (MPARAM)Request,               /* Ptr to request object */
                (MPARAM)Reply);                /* Ptr to reply object   */

     DosFreeMem(Request);                      /* Release request obj   */
     break;
       :
case WMP_REQUESTCOMPLETE:
     Reply=(PVOID)mp1;

     <Copy contents of Reply to private memory>

     DosFreeMem(Reply);                        /* Release reply object  */
     break;
```

Figure 102. Interprocess Communication Using Shared Memory (Part 1). This example shows a "requester" window procedure issuing requests and receiving replies by way of Presentation Manager messages.

If the technique described in 10.2, "Creating Another Process" on page 211 is followed, and the server process has posted a message to the client at the completion of server initialization, containing the window handle of the server's object window, the client can dispatch the request as an application-defined Presentation Manager message to the server's object window, with a pointer to the memory object as a message parameter. The server process then obtains access to the object using the **DosGetSharedMem()** function, as shown in Figure 103.

```
CASE WMP_DOREQUEST:
     Request = (REQUEST *)mp1;                  /* Get memory obj ptrs   */
     Reply = (REPLY *)mp2;
     DosGetSharedMem(&Request,                  /* Obtain access to obj  */
                     PAG_READ);                 /* Allow read access     */
     DosGetSharedMem(&Reply,                    /* Obtain access to obj  */
                     PAG_WRITE |                /* Allow write access    */
                     PAG_READ);                 /* Allow read access     */

     ServiceRequest(Request,Reply);             /* Complete request      */

     WinPostMsg(Request->hRequester,            /* Post msg to requester */
                WMP_REQUESTCOMPLETE,            /* Message class         */
                (MPARAM)Reply,                  /* Ptr to reply struct   */
                (MPARAM)0);

     DosFreeMem(Request);                       /* Free request object   */
     DosFreeMem(Reply);                         /* Free reply object     */
     break;
```

Figure 103. Interprocess Communication Using Shared Memory (Part 2). This example shows a "server" window procedure receiving and processing Presentation Manager messages.

In the simplest case where the client process has only one window, the handle of this window is passed to the server process when it is created, as part of the **DosExecPgm()** call, as shown in Figure 99 on page 212. Hence the server has access to the client's window handle and can pass the return data to the client. In a more complex situation where the client process has several windows and where a request can come from any of these, the handle can be passed as part of the request structure, as shown in Figure 102 on page 217 and Figure 103.

Another issue that arises when using shared memory to communicate between processes is that of freeing the shared memory object. When a process issues a **DosGiveSharedMem()** or **DosGetSharedMem()** call, the operating system increments a usage counter for the shared memory object, and will not release the memory until all processes using the object have issued a **DosFreeMem()** call.

For the server process that receives access to the shared memory object, the **DosFreeMem()** call is simply made whenever the server process has finished with the contents of the memory object. For the client process that initially creates the memory object, the **DosFreeMem()** call can be made at either of two points:

- If the client process does not care whether the request is correctly received by the server process, the **DosFreeMem()** call can be made immediately

after passing the message to the server, as shown in Figure 102 on page 217.

- If the client wishes to guarantee delivery of the request, it must pass the message, wait for an acknowledgement of receipt from the server, and then issue the **DosFreeMem()** call. This acknowledgement may simply be an indication of receipt, prior to the server processing the request, or may be the returned data from the request.

Note that it is common for returned data from a request to be passed using the same memory object as was used to contain the original request. In such cases, the memory object cannot be freed by the client process until the returned data is received and processed.

10.6.3 Atoms

This example assumes that the two processes described in the previous example require only to pass character strings, perhaps containing the request and the returned information. In this case, the requester obtains the handle to the system atom table using the **WinQuerySystemAtomTable()** function. It may then add the request string to this table using the **WinAddAtom()** function, and obtain an atom that represents the string in the table. This atom may then be passed to the server process in an application-defined Presentation Manager message. An example of this technique is shown in Figure 104 on page 220.

```
CASE WMP_SENDREQUEST:
     hSysAtomTable = WinQuerySystemAtomTable(); /* Get atom table handle */
     ReqAtom = WinAddAtom(hSysAtomTable,        /* Add string to table   */
                          szRequest);           /* String to be added    */
     WinPostMsg(hServer,                        /* Post msg to server    */
                WMP_DOREQUEST,                  /* DO_REQUEST message    */
                (MPARAM)ReqAtom,                /* Atom to access string */
                (MPARAM)hWnd);                  /* Return window handle  */

     <Store ReqAtom in window words>

     return((MRESULT)0);                        /* Return zero           */
     break;
       :
       :
case WMP_REQUESTCOMPLETE:
     hSysAtomTable = WinQuerySystemAtomTable(); /* Get atom table handle */
     ReplyAtom = (ATOM)mp1;                     /* Get atom for reply    */
     WinQueryAtomName(hSysAtomTable,            /* Get string from atom  */
                      ReplyAtom,                /* Atom                  */
                      szReply,                  /* Buffer for string     */
                      sizeof(szReply));         /* Size of buffer        */

     <Verify reply is correct>

     WinDeleteAtom(hSysAtomTable,               /* Delete atoms          */
                   ReqAtom);
     WinDeleteAtom(hSysAtomTable,
                   ReplyAtom);
     return(TRUE);
     break;
```

Figure 104. Interprocess Communication Using Atoms (Part 1). This example shows a "requester" window procedure issuing requests and receiving replies by way of Presentation Manager messages.

When the server process receives this message, it may also obtain the handle to the system atom table, and retrieve the string using the atom supplied in the message. If the string is of a predefined length, the server may simply retrieve the string using the **WinQueryAtomName()** function. If the string is of variable length, the server may need to obtain the length of the string using the **WinQueryAtomLength()** function, and allocate a buffer for the string. This is illustrated in Figure 105 on page 221.

```
CASE WMP_DOREQUEST:
     hAtomTable = WinQuerySystemAtomTable();     /* Get atom table handle */
     ReqAtom = (ATOM)mp1;                        /* Get atom for request  */
     hRequester = (HWND)mp2;                     /* Get requester handle  */

     ulLength = WinQueryAtomLength(hAtomTable,   /* Get size of string    */
                                   ReqAtom);
     szRequest = malloc(ulLength);               /* Allocate buffer       */

     WinQueryAtomName(hSysAtomTable,             /* Get string from atom  */
                      ReqAtom,                   /* Atom                  */
                      szRequest,                 /* Buffer for string     */
                      sizeof(szRequest));        /* Size of buffer        */

     ServiceRequest(szRequest,szReply);          /* Complete request      */

     ReplyAtom = WinAddAtom(hSysAtomTable,       /* Add string to table   */
                            szReply);            /* String to be added    */

     WinPostMsg(hRequester,                      /* Post msg to requester */
                WMP_REQUESTCOMPLETE,             /* Message class         */
                (MPARAM)ReplyAtom,               /* Atom to access string */
                (MPARAM)0);                      /* Return window handle  */

     free(szRequest);                            /* Free request buffer   */
     return((MRESULT)0);                         /* Return zero           */
     break;
```

Figure 105. Interprocess Communication Using Atoms (Part 2). This example shows a "server" window procedure receiving and processing Presentation Manager messages.

The server may return information to the requester using the system atom table. The **WinAddAtom()** function is used by the server to add the result string to the atom table, and the **WinQueryAtomLength()** and **WinQueryAtomName()** functions are used by the requester to retrieve the string. In the example shown in Figure 104 on page 220, it is assumed that the reply string returned to the requester is of a predefined length, and the **WinQueryAtomLength()** function is thus not required.

Note that the request string is not removed from the atom table until the server process has returned the result to the requester. Once the result is obtained and verified, the requester removes both the request and the result using the **WinDeleteAtom()** function.

The functions used to manipulate atoms and atom tables are described in detail in the *IBM OS/2 Version 2.0 Presentation Manager Reference.*

10.6.4 Queues

A queue can be used to pass information between the process that created it and other processes in the system. The process that creates the queue is known as the owner of the queue, and is the only process that can read and remove elements from the queue. Other processes may only add elements to the queue.

In a client-server environment, a queue is typically created by the server process. If a requester wishes to receive returned data from the server

process, it must therefore create its own queue. Figure 106 on page 222 shows a requester process using the **DosCreateQueue()** function to create its own queue, as well as using a queue owned by a server process.

```
#define SRVQUEUENAME = "\\QUEUES\\SRV_QUEUE"    /* Server queue name     */
#define REQQUEUENAME = "\\QUEUES\\REQ_QUEUE"    /* Requester queue name  */

HQUEUE      hReqQueue, hSrvQueue;               /* Queue handles         */
REQUESTDATA Server;                             /* Control information   */
REQUEST     *Request;                           /* Request data buffer   */
REPLY       *Reply;                             /* Reply data buffer     */
BYTE        Priority;                           /* Priority information  */
ULONG       ulBytes;                            /* Bytes read/written    */
APIRET      rc;                                 /* Return code           */

case WMP_SENDREQUEST:
     rc = DosCreateQueue(&hReqQueue,            /* Create req queue      */
                         QUE_FIFO  |            /* First-in, first-out   */
                         QUE_CONVERT_ADDRESS,   /* Convert addresses     */
                         REQQUEUENAME);         /* Name of queue         */

     rc = DosOpenQueue(&pidServer,              /* Open srv queue        */
                       &hSrvQueue,              /* Queue handle          */
                       SRVQUEUENAME);           /* Server queue name     */

     rc = DosAllocSharedMem(&Request,           /* Allocate shared mem   */
                            NULL,               /* object for request    */
                            sizeof(REQUEST),    /* Size of memory object */
                            PAG_WRITE |         /* Allow write access    */
                            PAG_READ  |         /* Allow read access     */
                            PAG_COMMIT);        /* Commit storage now    */
     rc = DosGiveSharedMem(Request,             /* Give mem to server    */
                           pidServer,           /* Server process id     */
                           PAG_READ);           /* Allow read only       */

     rc = DosWriteQueue(hSrvQueue,              /* Add request to queue  */
                        (ULONG)hWnd,            /* Requester win handle  */
                        sizeof(REQUEST),        /* Size of request       */
                        Request,                /* Request buffer        */
                        0);                     /* No priority           */
     rc = DosCloseQueue(hSrvQueue);             /* Close srv queue       */
     DosFreeMem(Request);                       /* Free request buffer   */
     break;
```

Figure 106. Interprocess Communication Using Queues (Part 1). This example shows elements being added to a queue by a "requester" process.

Queues may be created with a number of different ordering mechanisms. The order of elements in a queue is determined when the queue is created, and may be specified as FIFO, LIFO, or priority-based. When adding an element to a priority-based queue, a process must specify a priority (from 0 to 15 with 0 being lowest and 15 being highest) for the element.

Processes other than a queue's owner may gain write access to the queue using the **DosOpenQueue()** function to obtain a queue handle. Once this handle is obtained, the process may use the **DosWriteQueue()** function to add elements to the queue. The example in Figure 106 shows a requester process that

passes a request to a server process using a queue created by that server process.

The requester process first creates its own queue for returned data, using the **DosCreateQueue()** function. This queue will be accessed by the server to write the returned data from the completed request, which can then be read by the requester. The example shown in Figure 107 on page 224 creates a FIFO queue with the name specified in the string constant REQQUENAME, and specifies that the addresses of any elements placed in the queue by 16-bit processes are to be automatically converted to 32-bit addresses by the operating system. This conversion, specified using the QUE_CONVERT_ADDRESS flag, is used by 32-bit queue owners to avoid the need for the queue owner to explicitly convert addresses.

The requester obtains access to the server's queue using the **DosOpenQueue()** function, passing the queue name as a parameter. This function returns both a queue handle and the process identifier of the server process that owns the queue.

The requester must allocate a shared memory object to contain the request; the actual queue element contains only a pointer to that memory object. The requester then invokes the **DosGiveSharedMem()** function to provide read-only access (by specifying only the PAG_READ flag) to that object for the server process, using the process identifier returned by the **DosOpenQueue()** function.

The requester adds its request as a element in the queue, using the **DosWriteQueue()** function. Note that the second parameter to the function is an unsigned long integer, which may be used to pass application-specific information. The value specified in this parameter is passed to the queue owner as the *ulData* field of a REQUESTDATA structure, which is returned by the **DosReadQueue()** function. In this example, the parameter is used to pass the window handle of the requester's object window to the server process, so that a notification message can be passed to the requester when the request has been completed.

Once the element has been written to the queue, the requester immediately relinquishes access to the server's queue by issuing a **DosCloseQueue()** function call. The shared memory object allocated for the request buffer is then released by the requester using the **DosFreeMem()** function.

The server process is very similar in structure to the requester, in that it creates its own queue, then awaits and services requests. The server process is illustrated in Figure 107 on page 224.

```
#define SRVQUEUENAME = "\\QUEUES\\SRV_QUEUE"    /* Server queue name     */
#define REQQUEUENAME = "\\QUEUES\\REQ_QUEUE"    /* Requester queue name  */

HQUEUE      hSrvQueue, hReqQueue;               /* Queue handles         */
REQUESTDATA Requester;                          /* Requester win handle  */
REQUEST     *Request;                           /* Request data buffer   */
REPLY       *Reply;                             /* Reply data buffer     */
BYTE        Priority;                           /* Element priority      */
ULONG       ulBytes;                            /* Bytes read/written    */
APIRET      rc;                                 /* Return code           */

rc = DosCreateQueue(&hSrvQueue,                 /* Create queue          */
                    QUE_FIFO  |                 /* First-in, first-out   */
                    QUE_CONVERT_ADDRESS,        /* Convert addresses     */
                    SRVQUEUENAME);              /* Name of queue         */
while (!ProcessEnded)                           /* Until process ends    */
      {
      rc = DosReadQueue(hSrvQueue,              /* Read queue            */
                        &Requester,             /* Control information   */
                        &ulBytes,               /* Bytes read            */
                        &Request,               /* Data buffer pointer   */
                        0,                      /* Get first element     */
                        DCWW_WAIT,              /* Wait synchronously    */
                        &Priority,              /* Priority of element   */
                        0);                     /* No event semaphore    */

      ServiceRequest(Request);                  /* Process request       */

      rc = DosOpenQueue(&Requester.idpid,       /* Open queue            */
                        &hReqQueue,             /* Queue handle          */
                        REQQUEUENAME);          /* Server queue name     */
      rc = DosAllocSharedMem(&Reply,            /* Allocate shared mem   */
                             NULL,              /* object for request    */
                             sizeof(REPLY),     /* Size of memory object */
                             PAG_WRITE |        /* Allow write access    */
                             PAG_READ  |        /* Allow read access     */
                             PAG_COMMIT);       /* Commit storage now    */
      rc = DosGiveSharedMem(Reply,              /* Give mem to requester */
                            &Requester.idpid,   /* Req process id        */
                            PAG_READ);          /* Allow read only       */
      rc = DosWriteQueue(hReqQueue,             /* Add request to queue  */
                         0L,                    /* No control info       */
                         sizeof(REPLY),         /* Size of reply         */
                         Reply,                 /* Reply buffer          */
                         0);                    /* No priority           */
      rc = DosCloseQueue(hReqQueue);            /* Close queue           */

      DosFreeMem(Request);                      /* Free request buffer   */
      WinPostMsg((HWND)Requester.ulData,        /* Post notification msg */
                 WMP_REQUESTCOMPLETE,           /* to requester window   */
                 (MPARAM)Reply,                 /* Reply buffer pointer  */
                 0);
      DosFreeMem(Reply);                        /* Free reply buffer     */
      }
```

Figure 107. Interprocess Communication Using Queues (Part 2). This example shows the creation of a queue and the processing of items from the queue by a "server" process.

Note that the server process *does not* use an object window. It simply accepts requests from its own queue, using the DCWW_WAIT flag to suspend itself in the **DosReadQueue()** call until an element becomes available in the queue. Once a request is complete, the server places the returned data on the requester's queue, extracts the window handle of the requester from the REQUESTDATA structure provided by the **DosReadQueue()** call, and posts a message to the requester indicating that the request is complete. This message is processed by the requester to retrieve the returned data from the queue.

After adding the request to the server's queue, the requester is notified by the server when the request has been serviced. This is done using a Presentation Manager message, since the requester's window handle is passed to the server in the second parameter to the **DosWriteQueue()** function. The operating system imbeds this value as the second doubleword of a REQUESTDATA structure which is passed to the server by the **DosReadQueue()** function.

Once notification is received from the server process, the requester uses the **DosReadQueue()** function to retrieve the returned data from its own queue, as shown in Figure 108.

```
case WMP_REQUESTCOMPLETE:
     rc = DosReadQueue(hReqQueue,              /* Read req queue        */
                  &Server,                     /* Control information   */
                  &ulBytes,                    /* Bytes read            */
                  &Reply,                      /* Data buffer pointer   */
                  0,                           /* Get first element     */
                  DCWW_WAIT,                   /* Wait synchronously    */
                  &Priority,                   /* Priority of element   */
                  0);                          /* No event semaphore    */

     < Process reply>

     DosFreeMem(Reply);
     break;
```

Figure 108. Interprocess Communication Using Queues (Part 3). This example shows returned data being read from a queue by a "requester" process.

The DCWW_WAIT flag causes the **DosReadQueue()** function to wait until an element is available in the queue before returning control to the application. If the process merely wishes to check whether a queue element is available, the DCWW_NOWAIT flag may be specified, in which case an event semaphore must be created and its handle passed to the **DosReadQueue()** function. This semaphore is immediately set by the operating system, and is posted when an element is added to the queue. If the queue is shared between processes (as in the examples given herein), the semaphore must be shared, either by creating it as a named semaphore or by setting the DC_SEM_SHARED flag in the **DosCreateEventSem()** call.

It will be noted that the use of queues is very similar to that of shared memory, except that the queue is used to pass a pointer to a shared memory object, rather than a Presentation Manager message. However, queues have an advantage in that they may be FIFO, LIFO or priority-based, without the need for the application to handle the querying and sorting of elements.

10.6.5 Pipes

While a queue may only be read by its owner, a pipe may be used for either read access, write access or both. Pipes function in a similar way to files, and once created, are accessed using the OS/2 file system programming functions such as the **DosRead()** and **DosWrite()** functions. This means that pipes can be accessed by other applications in the system that support file system operations, including applications executing in virtual DOS machines. In this way, interprocess communication can be supported between an OS/2 application and a DOS or Windows application.

Pipes may be either named or anonymous. Communication via an anonymous pipe requires that the read and write handles for the pipe are known to both processes involved in the communication. Since these handles are not shared by default, another means of passing the handles, such as Presentation Manager messages or shared memory, must be used. For this reason, anonymous pipes are typically less useful than named pipes for interprocess communication, and are therefore used mainly for "streaming" communication between threads in the same process.

Named pipes are even more similar to files than anonymous pipes, since they are initially accessed using predefined names rather than requiring handles. Hence a process may easily obtain access to a named pipe, provided it knows the name of the pipe. Once the pipe has been created or opened, the process uses a pipe handle, which is similar to a file handle, to access the pipe via **DosRead()** and **DosWrite()** function calls.

OS/2 V2.0 introduces a number of new functions for accessing named pipes, which simplify programming in the client-server environment. These are the **DosCallNPipe()** function and the **DosTransactNPipe()** function, both of which are explained in the following text.

Note that while queues allow many processes to access and write to the queue, a named pipe is typically a one-to-one connection; the creator of the pipe may interact with only one other process at a time, and that process must relinquish access to the pipe before another process may gain access. For this reason, pipes have some limitations when used in a client-server environment with many requesters being serviced by a single server, as will become evident from the following examples.

A named pipe is normally created by the server process, using the **DosCreateNPipe()** function. During creation of the pipe, the server specifies the type of access that is allowed for the pipe. The following types of access are valid:

- Inbound access (client to server)
- Outbound access (server to client)
- Duplex (both).

Since only one client process may have access to a named pipe at any given time, the requester must wait for the named pipe to become available, using the **DosWaitNPipe()** function.

Figure 109 on page 227 shows a secondary thread routine in a requester process, which is dispatched to make a request to a server process.

```
#define  NPIPE_NAME   "\\PIPE\\SRVPIPE"          /* Pipe name              */

void RequestThread(TRANS *Trans)                  /* Requester thread       */
{
     ULONG      ulBytes;                          /* Bytes read/written     */
     APIRET     rc;                               /* Return code            */

     rc = DosWaitNPipe(NPIPE_NAME,                /* Wait on named pipe     */
                       NP_WAIT_INDEFINITELY);     /* Wait indefinitely      */

     rc = DosCallNPipe(NPIPE_NAME,                /* Pipe name              */
                       Trans->Request,            /* Request buffer ptr     */
                       sizeof(REQUEST),           /* Size of buffer         */
                       Trans->Reply,              /* Reply buffer ptr       */
                       sizeof(REPLY),             /* Size of buffer         */
                       &ulBytes,                  /* No. of bytes read      */
                       10000);                    /* Timeout period         */

     WinPostMsg(Trans->hReturn,                   /* Notify calling window */
                WMP_REQUESTCOMPLETE,              /* Request is complete    */
                (MPARAM)Trans,                    /* Transaction structure */
                0);
     DosExit(0);
}
```

Figure 109. Interprocess Communication Using Named Pipes (Part 1). This example shows a secondary thread in a "requester" process, writing to and reading from a named pipe.

In the example shown in Figure 109, the requester dispatches a secondary thread that waits synchronously for the named pipe to become available. This thread accepts a pointer to a data structure that contains the request and reply buffers along with the window handle of the window that initiated the thread. This window handle is included so that a notification message can be posted to the window when the request is complete. Since a separate thread is dispatched for each request, the thread terminates when the reply is returned by the server. Hence no object window is necessary.

When the pipe becomes available, the requester opens the pipe, writes the request, reads the reply and then closes the pipe. These functions are all performed by a single call to the **DosCallNPipe()** function. This function actually opens the pipe using a **DosOpen()** function call, and writes the request to the pipe using the **DosWrite()** function. The reply is read using the **DosRead()** function and the pipe is closed using the **DosClose()** function. Use of the **DosCallNPipe()** function simplifies the application code by allowing the programmer to combine these operations into a single function call.

Once the reply is received from the server and the **DosCallNPipe()** function returns, the requester thread notifies the window from which the request was made, by posting a Presentation Manager message to it. The pointer to the transaction data structure initially passed to the thread is returned with the message, enabling the window procedure to easily differentiate this request from any others that may currently be active.

The server process creates the named pipe, as shown in Figure 110 on page 228, using the **DosCreateNPipe()** function.

```
#define  NPIPE_NAME   "\\PIPE\\SRVPIPE"          /* Pipe name              */

HFILE      hPipe;                                 /* Pipe handle            */
REQUEST    *Request;                              /* Request buffer         */
REPLY      *Reply;                                /* Reply buffer           */
ULONG      ulAction;                              /* Open action            */
ULONG      ulBytes;                               /* Bytes read/written     */
APIRET     rc;                                    /* Return code            */

rc = DosCreateNPipe(NPIPE_NAME,                   /* Create named pipe      */
                    &hPipe,                       /* Pipe handle            */
                    NP_ACCESS_DUPLEX,             /* Allow duplex access    */
                    NP_WAIT          |            /* Blocking mode          */
                    NP_TYPE_MESSAGE  |            /* Msg oriented pipe      */
                    NP_READMODE_MESSAGE,          /* Msg oriented read      */
                    0x01,                         /* Single instance only   */
                    sizeof(REPLY),                /* Outbound buffer size   */
                    sizeof(REQUEST),              /* Inbound buffer size    */
                    0);                           /* Default timeout value  */

while (!ProcessEnded)                             /* Until process ends     */
      {
      rc = DosConnectNPipe(hPipe);                /* Connect to requester   */
      rc = DosRead(hPipe,                         /* Read request           */
                   Request,                       /* Request buffer         */
                   sizeof(REQUEST),               /* Size of buffer         */
                   &ulBytes);                     /* No. of bytes read      */

      ServiceRequest(Request, Reply);             /* Complete request       */

      rc = DosWrite(hPipe,                        /* Write reply to pipe    */
                    Reply,                        /* Reply buffer           */
                    sizeof(REPLY),                /* Size of buffer         */
                    &ulBytes);                    /* No. of bytes written   */
      rc = DosDisConnectNPipe(hPipe);             /* Disconnect from req    */
      }
```

Figure 110. Interprocess Communication Using Named Pipes (Part 2). This example shows a "server" process creating and reading from a duplex named pipe.

Once the pipe is created, the server process makes the pipe available to a requester process by issuing a **DosConnectNPipe()** function call. This enables any requester processes currently waiting for the pipe to contend for ownership. The requester that claims ownership returns from its **DosWaitNPipe()** call, while other requesters continue to wait.

The server then uses the **DosRead()** function to retrieve a request from the pipe. Since blocking mode is selected in the **DosCreateNPipe()** call by specifying the NP_WAIT flag, the **DosRead()** call does not return until a request becomes available.

Once the read operation completes, the server process services the request and writes the returned data back to the pipe using the **DosWrite()** function. It then informs the requester that the request has completed, using a Presentation Manager message, and obtaining the window handle of the requester from the *Request* structure.

Note that the server process cannot make the pipe available to other requesters by issuing a **DosDisconnectNPipe()** call, until the current requester has completed retrieval of the information from the pipe. If this call is issued before the requester retrieves its returned data, the requester's **DosRead()** call will fail. The server ensures that correct synchronization is maintained by passing the completion message *synchronously* using the **WinSendMsg()** function. For this reason, and in order to ensure that user responsiveness is maintained, it is recommended that requesters interacting with named pipes should do so from within object windows created in secondary threads under the control of the application's primary process.

Once the pipe is made available once more, this cycle of operations continues for each request issued to the server. The server process is suspended within the **DosConnectNPipe()** call until a request is issued by a requester process.

Note that where a one-to-one relationship exists between the server and requester processes, the requester need not relinquish access to the named pipe between requests. In such situations, the named pipe would be opened by the requester using the **DosOpen()** function directly, and accessed using the **DosTransactNPipe()** function. This function combines the **DosWrite()** and **DosRead()** functions. When the secondary thread is terminated, it can relinquish access to the pipe using the **DosClose()** function.

10.7 Maintaining Synchronization

It is the responsibility of the application to ensure the appropriate level of synchronization between threads or process accessing resources. Assuming the convention suggested in 10.1.2, "Threads Without Object Windows" on page 210, it is only necessary to indicate when a secondary thread or process has completed processing a particular unit of work. This may be achieved in a number of ways:

- By having the secondary thread or process post a completion message to the calling window procedure before terminating
- By using an event semaphore in conjunction with the Presentation Manager timer facility
- By using the **DosWaitThread()** function in the case of threads
- By using the **DosWaitChild()** function in the case of processes.

While it is possible, when using object windows in secondary threads or separate processes, to ensure synchronization by using the **WinSendMsg()** call for synchronous processing of the target window procedure, this method is *not* recommended since it prevents the calling window procedure from processing additional user input, and is thus potentially in violation of SAA CUA guidelines. In addition, immediate invocation of a window procedure in this way may disturb the natural sequence of message processing and compromise the user's intention.

10.7.1 Presentation Manager Messages

Since a process in OS/2 owns data resources, window handles are available to any threads under the control of that process. It is therefore possible for a secondary thread to post a message to the window procedure that invoked it, advising that the secondary thread has completed its processing. The window procedure may then process the message and take appropriate action.

This technique may be used by secondary threads that use object windows and those which do not. It requires only that the secondary thread have addressability to the window handle of the window procedure that invoked it. This handle may be obtained directly from Presentation Manager, but it is recommended that the handle of the invoking window procedure is passed to the secondary thread upon invocation. This may be done in one of two ways:

- By including the handle in the *CtrlData* parameter of the **WinCreateWindow()** call if the secondary thread is using an object window. This also requires passing the handle as a parameter to the **_beginthread()** call used to create the secondary thread's main processing routine.
- By including the handle as a parameter to the **DosCreateThread()** call if using a secondary thread without an object window.

The second method described above is illustrated in Figure 111.

```
        :
case WMP_THREAD:                        /* Start secondary thread      */
     DosCreateThread(ThreadID,          /* Thread ID                   */
                     Thread,            /* Entry point for thread      */
                     (PVOID)hwnd,       /* Invoking window handle      */
                     0L,                /* Start immediately           */
                     4096);             /* Stack size for thread       */
     break;

case WMP_ENDOFTHREAD:                   /* Thread has completed        */

     <perform end-of-thread processing>

     break;
        :
        :
int cdecl thread(hReturn)               /* Thread routine              */
    HWND hReturn;                       /* Handle of calling window    */
{
    <Perform lengthy processing task>

    WinPostMsg(hReturn,                 /* Post message to caller      */
               WMP_ENDOFTHREAD,         /* Message class               */
               0,0);                    /* No parameters               */
    DosExit(EXIT_THREAD,                /* Terminate thread            */
            0L);
}
```

Figure 111. Synchronization Using Presentation Manager Messages

Where the two communicating windows are under the control of different processes, the window handles must be explicitly communicated from one to the other since by default, the window handle of a window in one process is not available to a window in another process. One technique for achieving this

communication involves passing the window handle of the first window when the second process is created, and having the second window return a message to the first window after initialization, containing the window handle of the second window. This technique allows both communication and synchronization between windows. An example is given in 10.2, "Creating Another Process" on page 211.

10.7.2 Timers and Semaphores

Another method of achieving synchronization between threads or processes involves the use of an event semaphore and the Presentation Manager timer facility. The timer facility may be used from within a window procedure to create and start a timer that periodically sends messages of class WM_TIMER to the window, at intervals specified by the window procedure when the timer is created.

In this case, the WM_TIMER message is used by the window procedure in the primary thread or process, to periodically check the state of an event semaphore that indicates whether the secondary thread or process has completed its processing. The secondary thread or process sets the event semaphore upon commencing its processing, and releases (posts) it upon completion. The primary thread or process queries the state of the semaphore to determine when the secondary thread or process has completed its processing.

Note that when using this technique for synchronization between processes (rather than between threads within the same process), the event semaphore must be created as a shared semaphore, either by giving it a name or by specifying the DC_SEM_SHARED flag when invoking the **DosCreateEventSem()** function.

An example of a secondary thread using this technique is shown in Figure 112.

```
int cdecl thread()
{
    ulResult = DosCreateEventSem("\SEM32\THREAD", /* Name of semaphore  */
                                 hSem,            /* Semaphore handle   */
                                 NULL,            /* Not used           */
                                 FALSE);          /* Set immediately    */

    <Perform lengthy processing task>

    usResult = DosPostEventSem(hSem);             /* Release semaphore  */
    DosExit(0);                                   /* Terminate thread   */
}
```

Figure 112. Synchronization Using an Event Semaphore (Part 1). This example shows the routine executing in the secondary thread.

Note that the event semaphore is created as a shared semaphore and named. A named semaphore is recommended since, if the secondary thread routine is placed in a dynamic link library for subsequent use by other applications, or the secondary thread executes in a separate process, the name of the semaphore may be included in the documentation for that library, enabling calling window procedures to access the semaphore using the **DosOpenEventSem()** function (see Figure 113 on page 232). Using this technique promotes code reusability.

Figure 113 on page 232 shows a window procedure using the **WinStartTimer()** function to start a timer, immediately after dispatching a secondary thread such as the one shown in Figure 112. This timer in this example will cause a WM_TIMER message to be passed to the window every 0.5 seconds (500 milliseconds).

```
case WMP_THREAD:
     usReturn = DosCreateThread(ThreadID,       /* Create thread          */
                                Thread,         /* Entry point for thread */
                                NULL,           /* No initialization data */
                                0L,             /* Start immediately      */
                                4096);          /* Stack size for thread  */
     WinStartTimer(hAB,                         /* Start timer            */
                   hwnd,                        /* Window to get WM_TIMER */
                   TID_THREAD,                  /* ID of timer            */
                   500);                        /* Period in milliseconds */
     break;
        :
case WM_TIMER:
     ulResult=DosOpenEventSem("\SEM32\THREAD", /* Get semaphore handle   */
                              hSem);            /* Semaphore handle       */

     ulResult=DosWaitEventSem(hSem,             /* Check semaphore state  */
                              0);               /* Immediate timeout      */

     if (ulResult!=ERROR_TIMEOUT)               /* Semaphore not set      */
        {
        <perform end-of-thread processing>      /* Thread has completed   */
        }

     ulResult=DosCloseEventSem(hSem);           /* Close semaphore        */
     break;
```

Figure 113. Synchronization Using an Event Semaphore (Part 2). This example shows the window procedure in the primary thread, periodically testing to determine whether the event semaphore has been released.

Since the primary thread or process must remain responsive to the end user and thus cannot wait indefinitely for the semaphore to be released, the Presentation Manager timer facility is used to generate periodic WM_TIMER messages to the invoking window procedure in the primary thread or process. Upon receipt of each WM_TIMER message, the window procedure checks the state of the semaphore, timing out immediately if the semaphore has not yet been released by the secondary thread or process. This technique is illustrated in Figure 113.

Note once again the use of a named shared semaphore, in order to reduce the level of interdependence between the primary and secondary threads/processes, thus facilitating the inclusion of the secondary routine into a dynamic link library for subsequent use by other applications.

10.7.3 DosWaitThread() Function

Where a secondary thread must complete its processing *and terminate* before the primary thread can continue, the primary thread may use the **DosWaitThread()** function to determine whether the secondary thread has terminated. This function is used in conjunction with the Presentation Manager timer facility, to periodically check whether the secondary thread has issued a **DosExit()** function call. An example of a secondary thread using this technique is given in Figure 114.

```
int cdecl thread()
{
    <Perform lengthy processing task>

    DosExit(EXIT_THREAD,                          /* Terminate thread   */
            0L);                                  /* Return code        */
}
```

Figure 114. Synchronization Using the DosWaitThread() Function (Part 1). This example shows the routine executing in the secondary thread.

When the secondary thread has been started, the window procedure in the primary thread stores the thread identifier in its instance data area (typically using window words), and uses the Presentation Manager timer facility to send periodic WM_TIMER messages to itself, as shown in Figure 115.

```
case WMP_THREAD:
     usReturn = DosCreateThread(ThreadID,      /* Create thread          */
                                Thread,        /* Entry point for thread */
                                NULL,          /* No initialization data */
                                0L,            /* Start immediately      */
                                4096);         /* Stack size for thread  */

     <Store ThreadID in instance data block>

     WinStartTimer(hAB,                        /* Start timer            */
                   hwnd,                       /* Window to get WM_TIMER */
                   TID_THREAD,                 /* ID of timer            */
                   50);                        /* Period in milliseconds */
     break;
        :
case WM_TIMER:

     <Get ThreadID from instance data block>

     ulReturn=DosWaitThread(ThreadID,          /* Check thread status    */
                            DCWW_NOWAIT);      /* Immediate timeout      */
     if (ulReturn==ERROR_THREAD_NOT_TERMINATED) /* Thread still running  */
        break;                                 /* Continue waiting       */
     else                                      /* else                   */
        <perform end-of-thread processing>     /* Thread has completed   */
     break;
```

Figure 115. Synchronization Using the DosWaitThread() Function (Part 2). This example shows the window procedure in the primary thread, periodically testing to determine whether the secondary thread has terminated.

Whenever it receives a WM_TIMER message, the window procedure retrieves the thread identifier from its instance data area and uses the **DosWaitThread()** function to determine whether the thread has terminated. If so, it performs the required processing. If the thread has not yet terminated, it immediately returns control to Presentation Manager. Note the use of the **DosExit()** function in Figure 114. This assumes that the processing performed by the routine does not use an object window, and does not call C run-time library functions. As mentioned earlier in this chapter, secondary threads without object windows are typically used to perform a single, lengthy task, and terminate upon completion of this task. Since they are able to use the **DosExit()** function and the completion of their task causes the termination of the thread, such threads are ideal candidates for use of the **DosWaitThread()** function. For situations where the progress of execution must be indicated to the primary thread, an event semaphore is more suitable.

As already mentioned, there is very little difference between the use of the **DosWaitThread()** function and the use of an event semaphore. Both are used in conjunction with the Presentation Manager timer facility and in fact, both use an event semaphore. The **DosWaitThread()** function avoids the need for the application to explicitly open and check the semaphore, since the **DosWaitThread()** function performs these operations transparently. However, while an event semaphore may be used to indicate any significant event during execution of a secondary thread, while the **DosWaitThread()** function can only signal termination of the thread. Hence the **DosWaitThread()** function is slightly less flexible than the explicit use of an event semaphore with the Presentation Manager timer facility.

10.7.4 DosWaitChild() Function

The **DosWaitChild()** function allows a thread within a process to wait upon the termination of an asynchronous child process, in a similar manner to the **DosWaitThread()** function. The **DosWaitChild()** function allows a thread to wait for the termination of a single child process, or the termination of an entire process tree (that is, a process and all of its descendants).

Note that only the calling thread in the parent process is suspended during a **DosWaitChild()** call. If the parent process has other threads, they will continue to be dispatched.

The **DosWaitChild()** function can also be used to check the termination status of a child process that has already terminated, provided that process was started with the EXEC_ASYNCRESULT flag specified in the **DosExecPgm()** call. The use of this flag causes OS/2 to store the result code from the child process, for future reference by a **DosWaitChild()** call.

An example of the **DosWaitChild()** function is given in Figure 116.

```
rc = DosWaitChild(DCWA_PROCESS,          /* Wait for this process only */
                  DCWW_WAIT,             /* Wait until termination     */
                  &ReturnInfo,           /* Returned info              */
                  &pidServer,            /* Returned process ID        */
                  pidServer);            /* Process id to wait on      */
```

Figure 116. DosWaitChild() Function. This example assumes that the DosExecPgm() call shown in Figure 99 on page 212 has already been executed.

Specifying the DCWA_PROCESS flag in the first parameter of the **DosWaitChild()** call causes the calling thread to wait only upon the specified process, and not upon its children (if any). If a thread is to wait upon the entire process tree, the DCWA_PROCESSTREE flag must be specified.

The DCWW_WAIT flag in the second parameter causes the calling thread to wait until the specified process has terminated. The DCWW_NOWAIT flag would cause the **DosWaitChild()** call to return immediately, without waiting for a child process to end. The DCWW_NOWAIT flag is typically used when checking the termination status of a child process that has already ended.

10.8 Preserving Data Integrity

Since data resources are owned by a process, rather than by threads within the process, multiple threads may have addressability to the same static data storage areas, and potential problems arise with regard to serialization of data access and maintenance of data integrity. Similarly when multiple processes have access to a shared memory object, it is the responsibility of the application to ensure the integrity of shared resources; neither OS/2 nor Presentation Manager provide any automatic methods of avoiding such problems. However, mechanisms are provided whereby the application developer may prevent problems from occurring. Some suggested techniques for private data are as follows:

1. For any data that is private to a thread, use local variables defined within the thread, or automatic storage assigned from the stack (because each thread has its own stack memory object, this data is automatically protected since no other thread has addressability to this area of memory).
2. For any data that is private a particular window (as distinct from the window class), create a memory object to store this data and place a pointer in the window words, as described in 6.5.4, "Instance Data and Window Words" on page 81.
3. Specific data areas may be used to contain data that is passed between threads or processes. If this data is only accessed in response to particular messages passed between the threads or processes, and if these messages are only generated at predefined points in the application's execution (such as on entry to and exit from a window procedure), it is relatively simple for an application to control access to these data areas.

   Static allocation of such data areas is permissible where the accessing routines reside and execute solely under the control of a single application. However, where such routines are placed in a library and accessed by multiple applications, the potential for data corruption through application error increases significantly, and dynamic data allocation prior to invoking a secondary thread or passing a request to another process should be considered to ensure the integrity of data areas.
4. For any code that will be placed in a DLL, it is important that a separate set of memory objects is created for the data of each process that will access the DLL. In order to ensure this, a DATA NONSHARED statement should be specified in the module definition file (see 14.2.1, "Module Definition File" on page 278).

Note that the above techniques apply to data shared between threads within a process; OS/2 provides a variety of mechanisms for dealing with data and

memory areas that are shared between processes. These techniques are described in the *IBM OS/2 Version 2.0 Control Program Reference.*

10.9 Client-Server Applications

In situations where an object window is created in a secondary thread to manipulate a data object such as a database, or to handle access to a remote device or system, it is often desirable to have a single object window performing the requested actions, in response to requests from multiple display windows. This follows the basic architecture of a client-server application, in accordance with the object-oriented rule of allowing access to a data object *only* from a single application object, and therefore implements the concept of encapsulation.

For example, a user may use different display windows to access different views of the same database. However, for reasons of efficiency and data integrity, the actual database access should be coordinated by a single object window, preferably in a secondary thread in case a database access request causes a lengthy search.

The question then arises of how the handle of the object window may be made available to multiple display windows. A number of options are available:

- The handle may be stored as a global variable. This is not recommended however, since global variables are open to inadvertent modification, and their use imposes programming restrictions with respect to variable names.
- Immediately after the object window is created, its handle may be passed to all display windows that require communication with the object window. However, if subsequent modification of the application introduces a new display window, additional modifications would be required to the module that created the object window. This increases the interdependence between application objects, and is therefore not recommended.
- The handle of the object window may be stored in the window words of the application's main window. The handle of this window is available to all windows in the application, by querying the application's switch entry (as shown in 6.6.5, "Identifying the Destination Window" on page 91). If the window words of the application's main window are used to store a pointer to a data structure, which in turn contains the handles of object windows and other items of a global nature, these items may be retrieved by window procedures when required.

The final method described above is therefore the recommended solution. Object windows that will perform "server" tasks on behalf of a number of display window "clients" should be created by the window procedure for the application's main window, immediately upon creation of the main window, and the handles of the object windows stored in a data structure accessed via the window words of the main window.

10.10 Summary

OS/2 allows multiple threads of execution to be initiated within an application. Each thread is regarded as a distinct unit by the operating system, and is scheduled independently of other threads and processes in the system. Each application has a primary thread, created when the application is started. An application may optionally create one or more additional threads, known as secondary threads.

In certain circumstances, an application may also create additional processes to perform some portion of the application's processing. The use of additional processes may be necessary where different portions of the application's processing must be isolated from one another. It is also useful for applications that exploit the Workplace Shell since by default, all Workplace Shell objects share the same process and are hence unprotected from one another. The use of multiple processes has performance implications due to additional system overhead and should thus be implemented with care.

Secondary threads and processes may contain object windows, which do not appear on the display screen but act as addresses to which messages may be sent in order to initiate application processing. An object window typically "owns" a data object such as a database or controls access to an external entity such as a remote system. Where the processing in response to an application event requires access to another data object, the use of object windows in a secondary thread is recommended.

Communication with an object window is performed in the normal way using Presentation Manager messages. With suitable programming conventions, the handle of the object window may easily be made available to the calling window in order for such messages to be posted.

A single object window may receive messages from multiple windows, and perform actions on its data object on behalf of those windows. This approach allows easy coordination of requests for access to a data object, enhancing data integrity and efficiency of access. Applications that use this technique follow the basic client-server architecture, within the boundaries of the Presentation Manager application model.

Where the scope of a long-running event is restricted to a single method within the current application object, a secondary thread or process may be initiated without an object window. In such cases, a subroutine is initiated in the secondary thread, and the thread terminates immediately upon exiting that subroutine.

Presentation Manager provides a number of methods by which an application can synchronize access to data objects and determine whether a secondary thread or process has completed processing an event. These methods involve the use of functions such as **DosWaitThread()** or **DosWaitChild()**, Presentation Manager messages and/or event semaphores.

The multitasking capabilities of OS/2 allow greater application responsiveness since lengthy application processing may be performed in an asynchronous manner, leaving the application's primary thread free to continue interaction with the end user. This also facilitates conformance with the Systems Application Architecture CUA guideline stipulating that an application should

complete the processing of an event and be ready to handle further user input within 0.1 seconds.

Chapter 11. Systems Application Architecture CUA Considerations

The Presentation Manager environment provides the application developer with a rich set of functions that enable tasks to be performed in a variety of ways. However, within the general Presentation Manager application model, there are a number of considerations which, if observed, enable the design and development of applications that comply more closely with the guidelines and emerging conventions of the Systems Application Architecture CUA component. This chapter will discuss some of the considerations involved in designing an object-oriented, CUA-conforming Presentation Manager application.

The Presentation Manager programming interface enables the creation and manipulation of windows in a variety of ways by an application developer. However, there is considerable value in adopting a series of standard practices for developing Presentation Manager applications, from the viewpoint of consistency and ease of maintenance, and to enhance the degree of conformance to SAA CUA guidelines. The following points provide some guidelines on the interpretation of CUA specifications, and outline some emerging trends in the development of applications that use the Presentation Manager user interface.

11.1 Standard Windows

Standard windows are used to display the contents of data objects. The contents of a data object may, in turn, be comprised of other objects. For example, if the data object is a directory on a workstation's fixed disk, its contents are files, which are themselves objects. Alternatively, a data object may contain information in the form of text or formatted data. Note that an object may be defined by the user during execution; for example, if the user is editing a text file and selects a block of text to be operated upon, then that block of text becomes the scope of the following series of actions, and is thus defined as an object.

By convention, the nature of a user's interaction with a standard window should be unformatted and modeless; a standard window is used to display objects or their contents, from which the user selects an object upon which to perform one or more actions. The exact sequence of actions performed by the user in the window should not be of concern to the application. If a modal or otherwise structured dialog with the user is required, the application developer should implement this dialog as a dialog box. For this reason, it is recommended that the use of control windows be confined wherever possible to dialog boxes only. An allowable exception to this rule is the instance where a standard window displays a list of objects; in this case, the client window may be created as a container window or listbox.

This is relatively simple for normal listboxes; however, for a listbox with special display requirements and which is therefore created with the style LS_OWNERDRAW, the application must subclass the frame window in order to intercept and process the WM_DRAWITEM messages which are sent to the listbox's owner (the frame) whenever a listbox item must be redrawn on the screen.

A standard window should normally be both sizable and movable on the screen, allowing the user to configure the visual appearance of the desktop to suit the tasks being performed. A standard window should therefore be created with the FCF_SIZEBORDER style attribute in order to generate a sizing border for the window. Similarly, the user should be able to maximize and minimize the window in order to more clearly display information, unless the logical requirements of the application scenario dictate otherwise; the standard window should thus also be created with the FCF_MINMAX style attribute. A window that is neither sizable nor able to be minimized or maximized is by definition an optimized window, and should be implemented using a dialog box.

A standard window should always possess a title bar, to indicate the nature of the window's contents, and to provide a "handle" for moving the window on the screen; the frame window should therefore be created with the FCF_TITLEBAR style attribute. For an application's main window, the text displayed in the title bar should be the same as that displayed in the OS/2 Window List entry for the application, and should follow the convention "Object Name - View."

For child windows containing objects or their contents, the window title should be the same as the name or identifier of the item in the parent window that caused the child window to be created. For instance, the selection of a "Customer List" entry in the main window of an "Address Book" application might cause the display of a child window containing a list of customers' names; the title of this window would be "Customer List - Details View." Since a standard window represents an object or group of objects, the title should always be a noun rather than a verb.

A standard window is created using the **WinCreateWindow()** or **WinCreateStdWindow()** functions. **WinCreateWindow()** creates the frame and client windows in separate steps, whereas **WinCreateStdWindow()** creates both in a single step. A standard window is typically created with the FCF_SIZEBORDER, FCF_SYSMENU, FCF_TITLEBAR, FCF_MINMAX and FCF_MENU style attributes specified for the frame window. See Figure 19 on page 77 for an illustration of the use of these style attributes.

If an icon and/or accelerator table will be associated with the window, the FCF_ICON and FCF_ACCELTABLE style attributes should be specified. The icon and accelerator table definitions will then be loaded from the specified resource file when the window is created.

If the application does not wish to explicitly size and position a frame window on the desktop, the FCF_SHELLPOSITION style attribute may be specified. Presentation Manager will then determine a default size and position for the window.

As an alternative to specifying all of the above attributes, the FCF_STANDARD attribute may be specified. This attribute is assumed if the **WinCreateStdWindow()** call or the **WinCreateWindow()** call for a frame window contains no control data.

11.2 The Menu Bar

The menu bar is a menu which, in conjunction with its associated pulldown menus, enables the user to select an action from a list of valid actions, to be applied to a selected object or group of objects displayed in a window. The following guidelines should be followed in the design and layout of the menu bar, for the purpose of consistency and ergonomic behavior:

- A menu bar is not required for fewer than six actions, unless these are actions specifically defined by CUA.
- The number of entries within an menu bar should be kept to a minimum; it is better to have a "deep" pulldown menu with many choices than to have a "broad" menu bar. This reduces the number of options displayed to the user at any one time.
- A menu bar must have a pulldown menu associated with it.
- A pulldown menu item always represents an action, and therefore should always be a verb.
- Where selection of a menu bar entry results in the display of a pulldown menu, the menu bar entry should indicate the generic nature of the group of actions contained in the pulldown menu. If these actions pertain to a particular object or object class, the name of the object or class may be used as part of the menu bar entry. In such a case, the menu bar entry may be either a verb or a noun.
- If multiple distinct groups of actions, pertaining to a single object or class, are contained within the same pulldown menu, they should be separated by a horizontal separator bar within the pulldown menu.
- A pulldown menu item that when selected, results in the display of a dialog box,[4] should have its text succeeded in the pulldown menu by an ellipsis (...) to indicate that the dialog continues.

The SAA CUA guidelines stipulate that each standard window has a menu bar, which contains actions specific to the object represented by that window. However, certain actions may be inapplicable at certain points during application execution. If an action is not necessarily applicable at all stages of processing or for all data objects displayed within a window, that action should remain in the menu bar or pulldown menu, but should be disabled (that is, made non-selectable) until the point at which the action is valid.

Selection of a valid menu bar or pulldown menu item should result in the immediate performance of the action, or the display of a dialog box to obtain necessary information before performance of the requested action may take place. Selection of an invalid menu bar or pulldown menu item will result in a "beep" and the continued display of the pulldown. The menu bar and pulldown menus are discussed in the *IBM Systems Application Architecture CUA Advanced Guide to User Interface Design* and the *IBM Systems Application Architecture CUA Advanced Interface Design Reference.*

Presentation Manager provides mechanisms to achieve the insertion/deletion and enabling/disabling of menu bar items, as explained in the following sections.

[4] Note that a dialog box is known as an action window under CUA'91. However, the term dialog box is used in most Presentation Manager documentation, and will be used throughout this document for consistency.

11.2.1 Inserting/Deleting Menu Bar Items

The insertion of an item is achieved by sending a message of class MM_INSERTITEM to the system menu (or to the menu bar or appropriate pulldown menu) using the **WinSendDlgItemMsg()** function. The menu item information is placed into a data structure of type MENUITEM, as shown in Figure 117.

```
hFrame = WinQueryWindow(hwnd,                  /* Current Window    */
                        QW_PARENT);            /* Parent            */

hMenu = WinWindowFromID(hFrame,                /* Get handle of     */
                        FID_MENU);             /* menu bar          */

MenuItem.iPosition = MIT_END;                  /* Item position     */
MenuItem.afStyle = MIS_TEXT;                   /* Item style        */
MenuItem.afAttribute = 0;                      /* No attributes     */
MenuItem.id = MI_OPENOBJECT;                   /* Item identifier   */
MenuItem.hItem = 0;                            /* No handle         */
MenuItem.hwndSubmenu = 0;                      /* No p'down handle  */

rc = WinSendDlgItemMsg(hMenu,                  /* Send message      */
                       MI_FILE,                /* to File pulldown  */
                       MM_INSERTITEM,          /* Message class     */
                       &MenuItem,              /* Pointer to item   */
                       szItemText);            /* Text of menu item */
```

Figure 117. Dynamically Inserting a Menu Bar Item

In the example shown above, an item is to be inserted into a standard "File" pulldown menu. The menu item information is placed into the data structure *MenuItem*, and the text to appear in the pulldown menu is contained in the string variable *szItemText*.

Note however, that the menu bar is a child of the frame window, and the pulldown menu is a child of the menu bar. In order to successfully pass the message, the handle of the frame window must be obtained, and used to obtain the handle of the menu bar window. This handle is then used in the **WinSendDlgItemMsg()** call, along with the window identifier MN_FILE, to send the MM_INSERTITEM message to the pulldown menu. The frame window handle is obtained using the **WinQueryWindow()** function, and the QW_PARENT attribute causes the function to return the handle of the client's parent (that is, the frame window).

Once the frame window's handle is obtained, the various attributes of the *MenuItem* structure are initialized. An MM_INSERTITEM message is then sent directly to the pulldown menu.

Deletion of an item is accomplished in a similar fashion using a message of class MM_DELETEITEM. Both message classes and the MENUITEM data structure are described in the *IBM OS/2 Version 2.0 Presentation Manager Reference.*

11.2.1.1 Inserting/Deleting Separators

Where more than one logical group of items is contained within a single pulldown menu, these groups should be divided by a *separator*, which is a horizontal bar appearing between the last item of one group and the first item of the next, in order to provide a visual indication of the group's distinct identities. A separator may be defined within the resource script file (see Chapter 9, "Presentation Manager Resources") or may be inserted and deleted dynamically by the application. This dynamic insertion/deletion is carried out in a similar manner to that already described for "normal" pulldown menu items. However, the *afStyle* field in the MENUITEM structure is specified as MIS_SEPARATOR, and the text of the item is set to NULL. In such cases, an item identifier is not required for the separator, although it is recommended since an identifier must be specified in order to delete the separator from the pulldown menu should this be necessary at a later time.

Deletion of a separator is achieved in exactly the same manner as that described for a normal pulldown menu item, using a message of class MM_DELETEITEM and specifying the identifier of the separator to be deleted.

11.2.1.2 Inserting/Deleting Pulldown Menus

The technique for dynamically inserting a pulldown menu or cascade pulldown into the menu bar is basically similar to that already described for inserting a menu bar or pulldown menu item. An example is given in Figure 118.

```
HWND hPulldown;
  :
hPulldown = WinCreateMenu(hFrame,              /* Create empty menu */
                          NULL);               /* template          */

MenuItem.iPosition = MIT_END;                  /* Item position     */
MenuItem.afStyle = MIS_TEXT;                   /* Item style        */
MenuItem.afAttribute = 0;                      /* No attributes     */
MenuItem.id = MN_OPENOBJECT;                   /* Item identifier   */
MenuItem.hItem = 0;                            /* No handle         */
MenuItem.hwndSubmenu = hPulldown;              /* No p'down handle  */

rc = WinSendDlgItemMsg(hMenu,                  /* Send message      */
                       MN_FILE,                /* to File pulldown  */
                       MM_INSERTITEM,          /* Message class     */
                       &MenuItem,              /* Pointer to item   */
                       szItemText);            /* Text of menu item */
```

Figure 118. Dynamically Inserting a Pulldown Menu

The difference lies in the fact that a pulldown menu or cascade pulldown requires a menu template to be reserved in memory in order to contain the items that will subsequently be inserted into the pulldown. This template is created using the **WinCreateMenu()** function, which returns a handle to the menu template. This handle is then used in the *hwndSubmenu* field of the MENUITEM structure.

11.2.2 Enabling/Disabling Items

Disabling an item is achieved using the **WinEnableMenuItem()** function.Figure 119 shows how to disable the menu bar item MI_VIEW.

```
hMenu = WinWindowFromID(hFrame,            /* Get menu bar handle      */
                        FID_MENU);

rc = WinEnableMenuItem(hMenu,              /* Menu bar handle          */
                       MI_VIEW,            /* Menu item identifier       */
                       TRUE);              /* Enable menu item           */
```

Figure 119. Disabling an Menu Bar/Pulldown Menu Item

Enabling an item is achieved with the same message class, but FALSE is specified as the last attribute in the **WinEnableMenuItem()** function call.

Note that the **WinEnableMenuItem()** function is new in OS/2 Version 2.0; previous versions of OS/2 required use of the **WinSendDlgItemMsg()** function to send an MM_SETITEMATTR message to the menu.

11.2.3 Indicating Selected Items

A pulldown menu may be used to display a list of options from which one or more items may currently be selected; for example, a pulldown menu might provide a list of fonts to be used by a word-processing or desktop publishing application. SAA CUA guidelines stipulate that in such a case, the currently selected item or items within the pulldown menu should be indicated by a check mark next to the item. Presentation Manager provides support for this convention by allowing a check mark to be displayed within the pulldown menu. This is achieved using the **WinCheckMenuItem()** function; Figure 120 shows how to place a check mark next to the item MI_OPTION1.

```
hMenu = WinWindowFromID(hFrame,            /* Get menu bar handle      */
                        FID_MENU);

rc = WinCheckMenuItem(hMenu,               /* Menu bar handle          */
                      MI_OPTION1,          /* Menu item identifier       */
                      TRUE);               /* Set check mark             */
```

Figure 120. Placing a Check Mark on a Pulldown Menu Item

Note that the **WinCheckMenuItem()** function is new in OS/2 Version 2.0; previous versions of OS/2 required the use of a **WinSendDlgItemMsg()** function call to send an MM_SETITEMATTR message to the menu.

If a checked item is selected for a second time, the check mark should be removed by the application. This is achieved by sending the same message to the menu bar, with FALSE specified for the last attribute in the function call.

Note that the use of a check mark in a pulldown menu provides an alternative to the use of radio buttons or check boxes in simple dialogs. See 11.3.4, "Use of Control Windows" on page 253 for further information. When a pulldown menu with a check mark is used to display more than one set of mutually exclusive items, each set should be separated by a horizontal bar in the pulldown menu. See Chapter 9, "Presentation Manager Resources."

11.3 Action Windows

Under the guidelines provided by CUA'91, a dialog box is known as an **action window**. Since this chapter discusses CUA guidelines, this term will be used herein when referring to the provisions made by CUA. The term "dialog box" will be used when referring to the implementation of these concepts under Presentation Manager.

Action windows used by an application may be either modal or modeless, although modeless windows are preferred under CUA. Both types of action window are defined as **optimized windows**; that is, they are created at a predefined optimal size for their function, and may not be resized by the user. However, the use of each type of action window is different, as explained below.

A dialog box is typically created using the Dialog Box Editor application supplied as part of the *IBM Developer's Toolkit for OS/2 2.0*, and is defined in a dialog template which is stored in a .DLG file and referenced from the application's resource script file. Dialog templates are fully described in the *IBM OS/2 Version 2.0 Application Design Guide*.

It is recommended that wherever possible, action windows should be created with a title bar. Since action windows typically appear as the result of a menu bar item being selected, the title bar should contain the name of the parent object, plus the same text as the menu bar item. This provides the user with a visual indication of the action which led to the action window being displayed.

As already mentioned, an action window should not be sizable by the user, although it may be movable. Similarly, a modal action window should not include minimize or maximize icons, since the user must complete the interaction with the action window at its optimal size prior to continuing with execution. In the case of a modeless action window, there is no point in providing a maximize icon since the window is created at an optimal size for the information it will contain. However, since a user may wish to suspend the dialog in order to interact with other windows, the user may wish to remove the action window from the desktop. In such cases, a modeless action window should include a minimize icon.

11.3.1 Modeless Action Windows

A modeless action window is preferred in situations where the dialog with the user need not be completed before other user interaction with the application may occur. For instance, in an application object that performs an administrative procedure, data entry would typically be performed by the use of control windows. Since the control windows should remain displayed at all times, their parent window should not be sizable. By definition, the parent window is an optimized window, and should therefore be created as a dialog box. However, it may not be mandatory for the user to complete the dialog before interacting with other windows, and the dialog box should therefore be modeless.

Since the **WinDlgBox()** function automatically creates and executes a modal dialog box, a modeless dialog box must be created in one of two alternative ways:

- Using the standard **WinCreateWindow()** function, with the FCF_BORDER frame creation flag set. Control windows such as entry fields and buttons may then be created as children of the dialog box window.
- Using a **WinLoadDlg()** call to load the dialog template from a resource into memory, and then using the **WinShowWindow()** or **WinSetWindowPos()** functions to make the dialog box visible. The dialog template should have the FCF_BORDER attribute set. Control windows within the dialog box are defined in the dialog template.

The latter method is recommended for reasons of simplicity, since the dialog box and its control windows may be defined and stored in a resource file (see Chapter 9, "Presentation Manager Resources"), making the definition of the dialog box easier for the application developer.

The dialog box may be explicitly positioned on the screen, regardless of the method used. With the former method using the **WinCreateWindow()** function, the dialog box is positioned at the time it is created. With the latter method, the dialog box is positioned during its processing of the WM_INITDLG message.

The FCF_BORDER attribute results in the dialog box being displayed with a thin blue line as the dialog border. This is in accordance with the SAA CUA guidelines for modeless action windows.

Note that the two methods described above will result in different initialization messages being received. When created with the **WinCreateWindow()** function, the dialog box is regarded as a "normal" window, and a WM_CREATE message is passed to it. When created with the **WinLoadDlg()** function however, a WM_INITDLG message is passed instead. The application developer must bear this in mind when creating the dialog procedures for such dialog boxes.

11.3.2 Modal Action Windows

Modal action windows are used to carry out a dialog with the user in order to define or qualify the properties of a data object upon which the user is operating or wishes to operate. It is important to differentiate the properties of a data object from its contents; for instance, the properties of a text file might be its name, parent directory, archive/hidden/read-only attributes etc, whereas its contents would be the text within the file. Under Presentation Manager, manipulation of an object's contents is typically carried out in a standard window, whereas definition or alteration of attributes is done using a dialog box.

Note that where the dialog with the user is limited to a simple decision, a Presentation Manager message box should be used in preference to a dialog box for the implementation of an action window, since the coding effort and processing overhead associated with a message box is much less than that associated with loading and processing a dialog box. See 11.3.5, "Message Boxes" on page 256 for more information on the use of message boxes.

A modal dialog box is typically loaded and processed in a single step using the **WinDlgBox()** call. The modal nature of the dialog box is indicated to the user by a double blue line as the border for the dialog box, rather than the standard window border. The different border indicates that the dialog box is modal, and also indicates that it may not be sized by the user. This border is specified in the dialog template using the FCF_DLGBORDER attribute.

By default, a modal dialog box is **application-modal**; that is, the user must complete interaction with the dialog box before any further interaction may take place with windows in the current application. A dialog box may also be **system-modal**, in which case the user must complete interaction with the dialog box before interacting with *any* other window in the system. A system-modal dialog box is created in the same way as an application-modal dialog box, but with the additional attribute FS_SYSMODAL specified in the dialog template.

11.3.3 Standard Dialogs

OS/2 Version 2.0 provides standard dialog boxes for handling the selection of files and fonts. These dialogs conform to SAA CUA guidelines, and are implemented within Presentation Manager. Applications are therefore not required to explicitly design and code such dialog functions, nor to modify them should the CUA guidelines change in the future.

The standard dialogs are displayed using two Presentation Manager functions new to OS/2 Version 2.0; these are the **WinFileDlg()** and **WinFontDlg()** functions.

11.3.3.1 File Dialog

The standard file dialog enables a user to specify a file to be opened or a file name under which current work is to be saved, including the ability to switch directories and logical drives. The file dialog provides basic capabilities, and is designed in such a way that it may be modified if additional function is required.

The file dialog is displayed using the **WinFileDlg()** function. The dialog may be displayed as either an "Open" dialog or a "Save as" dialog, depending upon the value of control flags specified in a FILEDLG structure passed as a parameter to the function call. The **WinFileDlg()** function is shown in Figure 121 on page 248.

The appearance of the file dialog is controlled by the FDS_* style flags specified in the *fl* field in the FILEDLG structure. The fields in this structure are:

Field	Usage
cbSize	Defines the size of the FILEDLG structure, and should be initialized using the sizeof() function.
fl	Style flags of the form FDS_*, which control the attributes of the dialog. These flags are described in the *IBM OS/2 Version 2.0 Presentation Manager Reference*.
lUser	Used by applications to store their own state information if subclassing the dialog in order to modify its appearance or behavior.
lReturn	Identifier of the button used to dismiss the dialog. This is typically DID_OK or DID_CANCEL, unless the application has subclassed the dialog and added its own buttons.
lSRC	System return code which indicates the reason for dialog failure, if a failure has occurred. This field is used to assist in debugging.
pszTitle	Dialog title text. If set to NULL, the text will default to "Open" or "Save As", depending upon the FDS_* flags selected.
pszOKButton	Text used for the *OK* pushbutton on the dialog. If set to NULL, the text defaults to "OK".

```
USHORT OpenFile(HWND hOwner)
{
   extern PFNWP    WinFileDlg();          /* Function prototype          */
   extern FILEDLG  fild;                  /* File dlg control structure  */
   extern HFILE    hFileToOpen;           /* File handle                 */
   extern USHORT   usAction;              /* Action indicator            */
   static BOOL     fFirstTime = TRUE;     /* Flag                        */
   USHORT usReturn;                       /* Return code                 */

   if (fFirstTime)                        /* If invoked for first time   */
      {                                   /* build control structure     */
      fild.cbSize = sizeof(FILEDLG);      /* Set size of control struct  */
      fild.fl     = FDS_OPEN_DIALOG ]     /* Set dialog type to "Open"   */
                    FDS_CENTER      ]     /* Centered in parent window   */
                    FDS_HELP_BUTTON;      /* Include help button         */
      fild.pszTitle       = NULL;         /* Use default title bar text  */
      fild.pszOKButton    = NULL;         /* Use default button text     */
      fild.pfnDlgProc     = NULL;         /* Use standard dlg proc       */
      fild.hmod           = NULL;         /*  "     "      "    "         */
      fild.idDlg          = 0;            /*  "     "      "    "         */
      fild.pszIType       = NULL;         /* No initial type setting     */
      fild.ppszITypeList  = NULL;         /* No list of types            */
      fild.pszIDrive      = NULL;         /* No initial drive setting    */
      fild.ppszIDriveList = NULL;         /* No list of drivers          */
      fFirstTime = FALSE;                 /* Set flag to false           */
      }
   WinFileDlg(hOwner,                     /* Invoke file dialog          */
              &fild);                     /* Control structure pointer   */

   rc = DosOpen(fild.szFullFile,          /* Open returned file name     */
                &hFileToOpen,             /* File handle                 */
                &usAction,                /* Action indicator            */
                0L,                       /* File size not applicable    */
                0,                        /* File attribute ignored      */
                0x0001,                   /* Open file if it exists      */
                0x00C2,                   /* Non-shared, read-write      */
                0L);                      /* No sharing mode             */
   return(rc);                            /* Return                      */
}
```

Figure 121. Standard Dialogs - WinFileDlg() Function

pfnDlgProc Pointer to custom dialog procedure, for custom dialogs with the FDS_CUSTOM style flag set.

pszIType String pointer to a string defining the initial Extended Attribute type filter to be applied to the file name field in the dialog.

pszITypeList Pointer to a table of string pointers. Each points to a null terminated string defining an Extended Attribute type filter. The filters are displayed in ascending order in the Type pull-down box.

pszIDrive Pointer to a string specifying the initial logical drive to be applied in the dialog.

ppszIDriveList Pointer to a table of string pointers. Each points to a null terminated string defining a valid logical drive.

hMod If the FDS_CUSTOM style flag is set, this field defines the DLL module handle that contains the file dialog template to be used. If set to NULL, the dialog template is loaded from the application's EXE file.

szFullFile On initialization, this field contains the initial fully qualified path and file name, and on completion of the dialog, contains the selected or user-specified fully qualified file name. Upon invocation of the dialog, all drive and path data is stripped from the name, and moved to the appropriate fields in the dialog box.

ppszFQFilename Pointer to a table of pointers. Each points to a null terminated string containing a fully qualified file name. This table is used by applications that require multiple files to be selected from within the file dialog (indicated by specifying FDS_MULTIPLESEL). The storage is allocated by the file dialog procedure, and must be freed after dialog completion using the WinFileFreeFileList() function.

lFQFCount Number of file names selected in the file dialog, for dialogs with multiple selection enabled.

idDlg Window identifier of the dialog window. If the FDS_CUSTOM style flag is set, this is also the resource identifier of the dialog template.

x,y Position of the dialog, relative to its parent. These fields are automatically updated by the dialog procedure when the dialog is moved by the user, so that if the same FILEDLG procedure is used on subsequent invocations, the dialog will appear in the same location. The FDS_CENTER style flag overrides any settings specified.

sEAType Extended Attribute file type to be be assigned to the file. This field contains the returned value specified in the *Type* field in the dialog. This field is valid only for a "Save As" dialog; the value -1 is returned for an "Open" dialog.

For applications with specialized file handling requirements, the standard file dialog may be subclassed, allowing these requirements to be handled while retaining standard processing for the majority of events. This subclassing is invoked by specifying the address of an application-defined dialog procedure in the *pfnDlgProc* field in the FILEDLG structure, and by specifying the resource identifier of an application-defined dialog template if controls are to be added or removed from the dialog.

Note that application-defined dialog procedures should invoke the **WinFileDlgProc()** function as their default case for message processing, to ensure that messages not explicitly processed by the application are passed to the standard file dialog procedure for correct default processing.

11.3.3.2 Font Dialog

The standard font dialog enables a user to specify a choice of font names, styles, and sizes from the range available within a given application. The font dialog is intended to fit basic application needs, and is designed in such a way that additional function may be added by subclassing the dialog procedure.

The font dialog is displayed using the **WinFontDlg()** function, specifying the owner window for the dialog box, and a FONTDLG control structure. The use of the **WinFontDlg()** function is shown in Figure 122 on page 251.

The appearance of the dialog is determined by the FNTS_* flags specified in the *fl* field of the FONTDLG structure, and by the other fields in this structure. The fields in the FONTDLG structure are:

Field	Usage
cbSize	Defines the size of the structure, and should be initialized using the sizeof() function.
hpsScreen	If not NULL, this field specifies the screen presentation space, which the dialog procedure queries for available fonts.
hpsPrinter	If not NULL, this field specifies the printer presentation space, which the dialog procedure queries for available fonts.
pszTitle	Title text for the dialog box. If set to NULL, the default text "Font" is used.
pszPreview	Text to be displayed in the *Preview* field in the dialog box. If set to NULL, the default text "abcdABCD" is used.
pszPtSizeList	String containing a list of numeric point sizes, to be displayed in the *Point Size* drop-down list in the dialog box. Point sizes within the string must be separated by spaces. If set to NULL, the defaults of 8, 10, 12, 14, 18, and 24 are used.
pfnDlgProc	Pointer to custom dialog procedure, for dialogs with the FNTS_CUSTOM flag set.
szFamilyname	Font family name to be used by an application to select a font. If set to NULL, the system default is used.
fxPointSize	Vertical point size of the font.
fl	Flags which specify the characteristics of the dialog box; these may be any combination of FNTS_CENTERED, FNTS_CUSTOM, FNTS_HELPBUTTON, FNTS_MULTIFONTSELECTION, and FNTS_MODELESS. Flags are combined using the "or" operator.
flFlags	Flags; specifying FNTF_VIEWPRINTERFONTS specifies whether printer fonts should be included if both *hpsScreen* and *hpsPrinter* are non-NULL. FNTF_PRINTERFONTSELECTED is set upon return, if the user selects a printer font.
flType	Specifies additional font attributes specified by the user, and may be used as the options field in a QFSATTRS structure for the **GpiQueryFaceString()** function.
flTypeMask	Specifies which flags in the *flType* field are required to change. This is only relevant where selections may be for different types and styles when multiple fonts are being selected.
flStyle	Specifies any additional selections, and may be used for the selection indicators in a FATTRS structure supplied to the **GpiCreateLogFont()** function.
flStyleMask	Specifies which flags in the *flStyle* field are required to change. This is only relevant where selections may be for different types and styles when multiple fonts are being selected.

```
void SetFont(HWND hOwner, HPS hpsScreen, USHORT usCodePage)
{
     extern PFNWP   WinFontDlg();              /* Function prototype      */
     extern FONTDLG fntd;                      /* Dialog control struct   */
     static BOOL    fFirstTime = TRUE;         /* Flag                    */
     CHARBUNDLE     cbnd;                      /* Attributes              */

if (FirstTime)                                 /* If invoked for 1st time */
   {                                           /* build control structure */
   fntd.cbSize            = sizeof(FONTDLG);  /* Set size of structure   */
   fntd.fl                = FNTS_CENTER ]     /* Specify centered dlg    */
                            FNTS_HELPBUTTON;  /* Include help button     */
   fntd.hpsPrinter        = NULL;             /* No printer font         */
   fntd.pszTitle          = "Fonts";          /* Dialog title text       */
   fntd.pfnDlgProc        = NULL;             /* Use standard dlg proc   */
   fntd.hmod              = NULL;             /*  "     "     "    "     */
   fntd.idDlg             = 0;                /*  "     "     "    "     */
   fntd.pszPreview        = NULL              /* Default preview string  */
   fntd.pszPtSizeList     = NULL;             /* Default point sizes     */
   fntd.flFlags           = 0L;               /* Default flags           */
   fntd.szFamilyname[]    = '\0';             /* System default          */
   fntd.fxPointSize       = MAKEFIXED(12,0);  /* 12-point vertical size  */
   fntd.usWeight          = FWEIGHT_NORMAL;   /* Weight or thickness     */
   fntd.usWidth           = FWIDTH_NORMAL;    /* Character width         */
   fntd.flType            = 0L;               /* No additional attribs   */
   fntd.flStyle           = 0L;               /* No additional styles    */
   fntd.flCHSOptions      = 0L;               /* No additional options   */
   fntd.clrFore           = CLR_BLACK;        /* Black characters        */
   fntd.clrBack           = CLR_WHITE;        /* White background        */
   fntd.fAttrs.usCodePage = usCodePage;       /* Specified code page     */
   fFirstTime=FALSE;                           /* Reset flag              */
   }
fntd.hpsScreen=hpsScreen;                      /* Set presentation space  */

WinFontDlg(hOwner,                             /* Invoke font dialog      */
           &fntd);                             /* Control structure ptr   */

GpiCreateLogFront(hpsScreen,                   /* Create logical font     */
                  "Name    ",                  /* Name of font            */
                  0,                           /* Local font identifier   */
                  fntd.fAttrs);                /* Returned attributes     */
cbnd.lColor = fntd.clrFore;                    /* Set foreground color    */
cbnd.lBackColor = fntd.clrBack;                /* Set background color    */
GpiSetAttrs(hpsScreen,                         /* Set attributes          */
            PRIM_CHAR,                         /* Character attributes    */
            CBB_COLOR ] CBB_BACK_COLOR,        /* Attributes to be set    */
            0L,                                /* Defaults mask           */
            (PBUNDLE)&cbnd);                   /* Attribute structure     */
GpiCharStringPos(hpsScreen,                    /* Write character string  */
                 NULL,                         /* No rectangle            */
                 fntd.flCHSOptions,            /* Options                 */
                 4,                            /* Number of bytes         */
                 "Text",                       /* Text string             */
                 NULL);                        /* Increment values        */
}
```

Figure 122. WinFontDlg() Function - Sample Code

flCHSOptions These are equivalent to the CHS_* option flags used by the **GpiCharStringPos()** and **GpiCharStringPosAt()** functions.

flCHSMask Similar to *flStyleMask*.

clrFore Foreground color for the font.

clrBack Background color for the font.

lUser May be used by applications to pass information if subclassing the font dialog.

lReturn Identifier of the button used to dismiss the dialog. This is typically DID_OK or DID_CANCEL, unless the application has subclassed the dialog and added its own buttons.

lSRC System return code that indicates the reason for dialog failure, if a failure has occurred. This field is used to assist in debugging.

lEmHeight Value that may be used within a FONTMETRICS structure by applications.

lXHeight As above.

iExternalLeading
As above.

fAttrs Complete font attribute (FATTRS) structure for the selected font. Only the codepage field may be modified by the application prior to invoking the dialog.

sNominalPointSize
Nominal point size of selected font.

usWeight Character thickness (for example, normal or bold). The returned value may be used in the *weightclass* field in the QFSATTRS structure for the **GpiQueryFaceString()** function.

usWidth Character width. The returned value may be used in the *widthclass* field in the QFSATTRS structure for the **GpiQueryFaceString()** function.

x,y Position of the dialog relative to its parent. These fields are automatically updated by the dialog procedure when the dialog is moved by the user, so that if the same FONTDLG structure is used on subsequent invocations, the dialog will appear in the same location. The FNTS_CENTERED style flag overrides any settings specified.

idDlg Window identifier of the dialog window. If the FNTS_CUSTOM style flag is set, this is also the resource identifier of the dialog template.

hmod If the FNTS_CUSTOM style flag is set, this field defines the DLL module handle that contains the file dialog template to be used. If set to NULL, the dialog template is loaded from the application's EXE file.

Applications may customize the font dialog through subclassing, by specifying the FNTS_CUSTOM style flag, giving the resource identifier and module handle of the application's customized font dialog template, and the address of an application-defined dialog procedure, in the FONTDLG structure. The **WinFontDlg()** function then performs the subclassing operation on the application's behalf.

Note that application-defined dialog procedures should invoke the **WinFontDlgProc()** function as their default case for message processing, to ensure that messages not explicitly processed by the application are passed to the standard font dialog procedure for correct default processing.

An application that uses its own dialog template *must* include all of the standard controls within the dialog box, in addition to its own customized controls. Those controls, which are not required, may be rendered invisible in order to provide the correct appearance.

Control window identifiers in the range 0x0000 to 0x0FFF are reserved for use by standard controls. The application's own controls should therefore use window identifiers greater than 0x0FFF.

11.3.4 Use of Control Windows

A number of control window classes are provided by Presentation Manager. Under CUA guidelines, these control windows should be displayed in standard windows, although their use is more typically in dialog boxes. This is in accordance with the convention that windows display objects or the contents of objects, and other more structured information such as object attributes is displayed in a dialog box. See also the proviso regarding listboxes under 11.1, "Standard Windows" on page 239. The use of control windows is defined in the *IBM Systems Application Architecture CUA Advanced Guide to User Interface Design*. However, there are some emerging conventions as to the exact interpretation of the CUA guidelines, and these are discussed in the following sections.

11.3.4.1 Entry Field

Entry fields are used where textual or numeric data is required from the user, and where the set of possible entries is open-ended. Examples of such data items include file names, object descriptions etc.

11.3.4.2 List Box

A listbox is used to display a list of objects, where the contents of that list may change from one execution to the next, based upon various external or user-specified criteria. One or more items may be selected from a listbox. The listbox is typically created with a size sufficient to display a certain number of items, and a scroll bar is provided if the number of items increases such that not all items may be displayed at once.

Note that a listbox should not be used to display a set of choices where that set is finite and unchanging. In such a case, radio buttons may be used where the choices are mutually exclusive, or check boxes used where more than one choice may be made concurrently.

11.3.4.3 Combo Box

A combo box, also known as a *prompted entry field* is a combination of the entry field and listbox control window styles, and is supported by Presentation Manager in OS/2 Version 1.2 and above. A combo box is used where textual or numeric data is required from a user, where the set of possible entries is finite, and where the application wishes to prompt the user for a valid entry. A combo box may be of three distinct types:

- A **simple** combo box is displayed as an entry field with a listbox directly below it. The user may enter textual or numeric data into the entry field in

the same way as for a normal entry field, or may select an item from the listbox. The selected item will then appear in the entry field. The simple combo box provides a similar function to that of a single-selection listbox.

- A **drop-down** combo box is displayed as an entry field with an icon to its immediate right. The user may enter textual or numeric data into the entry field, or may select the icon. When selected, the icon causes a listbox to appear below the entry field, containing a list of valid entries for the entry field. When selected, an item appears in the listbox. The drop-down combo box is recommended where the set of valid entries is finite and limited, but where the user may already know the required entry and may wish to save time by entering it him/herself.
- A **drop-down list** combo box is displayed in a similar manner to a drop-down combo box, as an entry field with an icon to its immediate right. However, the user may not enter data directly into the entry field, but must select an item from the listbox. A drop-down list combo box is recommended in situations where a number of control windows are located in a dialog box, where the optimization of space is of primary importance, and where the default entry is likely to be used.

The use of a combo box is typically recommended in place of a listbox in order to save room within a dialog box, or in place of an entry field where the application wishes to prompt a user with a list of valid entries.

11.3.4.4 Radio Button

Radio buttons are used to indicate a group of mutually exclusive options; that is, only one of the items in the group is selectable at any one time, and selecting one item automatically deselects any previously selected item. Selecting a radio button does not complete the dialog; a user may revise his/her selection any number of times during the dialog. Once a final decision is made, the user completes the dialog using a push button (see below).

Radio buttons are always displayed in groups; it makes no sense to have a single mutually exclusive selection item. Text is displayed along with the buttons to indicate the choice represented by each button. If multiple groups of radio buttons are present within a dialog box or window, or if radio buttons are combined with other types of control window, it is recommended for reasons of clarity that the radio buttons be placed within a group box, and that this group box be named to indicate the nature or purpose of the group as a whole.

As described above, radio buttons should be used to denote a set of mutually exclusive options in the creation or manipulation of an object, as part of a more complex dialog. They should not be used to present a set of options in response to an application or system event, where this set of options is the sole purpose of the dialog. In such cases, a message box is the preferred mechanism to achieve this type of communication with the user, since the processing overhead associated with a message box is less than that associated with a dialog box. For example, a warning that the user is about to exit the application without saving his/her latest set of changes would be presented using a message box rather than a dialog box with radio buttons.

The equivalent function of a group of radio buttons may also be provided by a pulldown menu displaying a set of options, only one of which may be selected at any one time, with the selected item indicated by a check mark. The use of a pulldown menu is the recommended option in situations where the selection of an option is the only action to be performed. The use of radio buttons is

recommended where the selection of an option indicated by the radio buttons is part of a more complex dialog.

Note that from a programming viewpoint, "auto" radio buttons should be used in preference to standard radio buttons since these buttons are drawn and maintained by Presentation Manager. The application need not concern itself with redrawing the buttons when their state changes, thereby allowing simpler programming.

11.3.4.5 Check Box

A check box is used to indicate a single option that may be toggled on or off by the user. Multiple check boxes may appear in a single dialog box or window, and may refer to different attributes of the same object. However, these attributes are related to each other only by their application to that same object, and should not be mutually exclusive.

A **3-state** button is a special type of check box that, in addition to being marked selected or non-selected, may be "grayed out" to indicate that a choice is non-selectable in the current dialog. A 3-state button should be used whenever a dialog box is applicable to a range of objects, but where certain options within the dialog box are not valid for all objects dealt with by that dialog box. A 3-state button may be enabled or disabled using the **WinEnableWindow()** function, obtaining the window handle of the button from a **WinWindowFromID()** call.

The equivalent function to a check box may be provided by a pulldown menu displaying a list of options, from which multiple items may be selected at any one time, with the selected items indicated by check marks. The use of a pulldown menu is recommended where the selection of such an option is the only action to be performed. The use of a check box or 3-state button is the preferred solution where the selection is part of a more complex dialog.

Note that the "auto" versions of check boxes and 3-state buttons should be used in preference to the standard versions, since the auto versions are maintained by Presentation Manager, and the application need not concern itself with redrawing these buttons when their state changes.

11.3.4.6 Push Button

Push buttons are used to initiate an immediate action by the application. If desired, push buttons can be used in conjunction with the menu bar and context menu, to provide easy access to commonly used functions in both primary windows and action windows.

Push buttons should not be used to form menus of selectable options that cause child windows to appear when a push button is selected. Such a practice effectively forms a hierarchical user interface, which is in violation of object-oriented user interface principles.

An exception to this rule is a "Help" push button, which immediately displays a window containing help information, while maintaining the previous window or dialog on the screen. Dismissing the help window returns the user to the original window in which the "Help" push button was displayed.

11.3.4.7 Slider

The slider is used where a single value must be selected from a continuous range of options. For example, the brightness of the screen, the saturation of a color or the speed of the mouse cursor on the screen are all values selected from a continuous, though finite, range of options.

Under previous versions of OS/2, scroll bars were often used to provide a portion of the slider's functionality. The provision of the slider control under OS/2 Version 2.0 allows the scroll bar to be used *only* for its intended purpose of scrolling information within a window; this improves the consistency of the user interface and removes a potential source of user confusion.

11.3.4.8 Value Set

The value set control is used in a similar way to a set of radio buttons, to indicate a group of mutually exclusive options. Many of the comments made for radio buttons apply equally to value sets.

However, while the use of radio buttons is effectively limited to text items, a value set allows the use of text or graphical items, as well as color patches. Thus a value set provides additional flexibility where a selection must be made from a set of mutually exclusive options, providing a mechanism for the display of those options to the user, and allowing the user to directly select the required choice.

11.3.5 Message Boxes

By convention, message boxes are used to inform the user of an event, and to carry out a dialog with the user in the following circumstances:

- Where the information to be conveyed to the user is simple and limited to no more than a few lines of text
- Where the input to be gained from the user is limited to a single decision from a limited list of mutually exclusive options.

The type of decision available to the user from a message box is also limited to simple choices such as "Yes/No," "OK/Cancel," or "Yes/No/Help." Since the buttons displayed in the message box are push buttons, selecting any button will result in the removal of the message box from the screen and immediate action on the part of the application. An exception to this rule is the "Help" button, which should result in the message box being left on the screen, and the simultaneous display of a window containing help information. This help window should be a child of the message box, so that it is dismissed when the message box is closed, and the input focus should return to the message box when the help window is dismissed.

A message box is created and processed using the **WinMessageBox()** function as follows:

```
rc = WinMessageBox(hDesktop,                /* Desktop is parent     */
                   hwnd,                    /* Curr. window is owner */
                   szMessageText,           /* Message text          */
                   "Warning",               /* Title of message box  */
                   0,                       /* No message box ident. */
                   MB_YESNO |               /* Yes/No choices        */
                   MB_DEFBUTTON1 |          /* Yes is default choice */
                   MB_CUAWARNING);          /* Warning style         */
```

Message boxes may be of three types:

- **Notification** message boxes inform the user of a system event that requires his/her attention, but does not signify an error or potential error condition. Such message boxes are created with the message style MB_CUANOTIFICATION.
- **Warning** message boxes inform the user of a potential error condition that may affect the integrity of the application or its data; for example, the user may try to exit the application without saving the latest set of changes to a data object. Such message boxes are created with the message style MB_CUAWARNING.
- **Critical** message boxes inform the user of an error condition that requires his/her immediate attention; for example, a diskette may be unreadable. Such message boxes are created with the message style MB_CUACRITICAL.

These message box styles may be combined with other style identifiers that determine the buttons to be displayed in the message box, the default action and the modality of the message box (that is, system- or application-modal). These identifiers are described, along with the **WinMessageBox()** function, in the *IBM OS/2 Version 2.0 Presentation Manager Reference*.

A message box should always be created with a title; while Presentation Manager provides a default message title if a null title string is specified in the **WinMessageBox()** function, the use of this feature is not recommended. A message box title should identify the object or action from which the message originated; for example, a message originating from a "Parts List" object would have the title "Parts List," whereas a message occurring as a result of an incorrectly completed dialog during an "Open File" action would have a title of "Open File."

A message box is created as an optimized window, and by default is neither moveable nor sizable. However, the situation may arise where the message is related to some information displayed in a window, and the decision to be made by the user is dependent upon the nature and context of that information. The user must therefore be able to view the information in order to make the decision required by the message box, and the information may be hidden by the message box itself. To avoid this situation, a message box may be specified with the style identifier MB_MOVEABLE, which allows the message box to be moved (although not sized) by clicking the mouse on its title in a similar fashion to that used for a standard window.

11.4 Maintaining User Responsiveness

The particular implementation of the message handling concept under Presentation Manager means that it is possible for a user to be "locked out" of the system if a window procedure does not return from processing a message within a reasonable period of time, since Presentation Manager only dispatches messages from the system queue to other applications when the currently active application attempts to retrieve a message from its queue.

The "reasonable" period of time is defined under Presentation Manager guidelines to be 0.1 seconds; a window procedure should complete the processing of a message within this period and return control to the application's main routine in order that the main routine may issue its next **WinGetMsg()** call. While this time period is adequate for processing of most

messages, it may be insufficient for some messages; for examples, those that involve lengthy operations such as access to a remote system.

In order to avoid the situation where an application or the entire Presentation Manager environment is unresponsive to the end user, OS/2 allows applications to create secondary threads of execution to handle lengthy processing, thus enabling the application's primary thread to continue responding to user interaction. Presentation Manager also provides a number of methods by which synchronization between threads may be achieved in order to ensure the integrity of the user's intention and of data resources manipulated by the application. The use of secondary threads, and techniques for synchronizing execution between threads and processes, are discussed in Chapter 10, "Multitasking Considerations."

11.5 Summary

It can be seen that there are emergent programming conventions governing the use of Presentation Manager and operating system constructs to implement CUA-conforming, object-oriented applications under the Presentation Manager application architecture. These principles facilitate the achievement of the many benefits attributable to the use of an object-oriented design approach, while remaining within the framework of Systems Application Architecture.

Guidelines have been established, in accordance with emerging conventions, with regard to the use of standard and control windows, dialog boxes and message boxes, and the nature of their interaction with the user. Each type of window has its own particular role in the interaction between the application and the end user; adherence to these conventions will provide a greater level of consistency between applications, within the parameters of the CUA specifications.

The CUA guidelines specify that an application should have a response time of no more than 0.1 seconds, after which time the application should be ready to process the next user interaction. However, the situation may arise where a unit of processing takes longer than the allowed time period. A number of ways exist by which the application may overcome this problem, typically involving the use of asynchronous threads, either with or without object windows.

The embodiment of these principles into Presentation Manager applications at the design stage results in closer conformance to Systems Application Architecture guidelines. The enterprise may thereby achieve the benefits of a consistent and intuitive approach to the human-machine interface between users and applications.

Chapter 12. Application Migration

OS/2 Version 2.0 provides application compatibility at the executable code level for applications written to run under previous versions of OS/2. Hence no modification is necessary to enable such applications to run under Version 2.0. However, in order to take full advantage of the enhanced capabilities of OS/2 Version 2.0, such as the 32-bit flat memory model and the additional features implemented by Presentation Manager, applications will require modification of their source code.

Application developers who wish to migrate their code to the 32-bit programming environment under OS/2 Version 2.0 should experience little difficulty. Changes between the OS/2 Version 2.0 programming environment and that provided under previous versions of OS/2 fall into the following basic categories:

- Data types
- Function name changes
- Function enhancements
- Memory management
- New Presentation Manager functions.

The remainder of this chapter will describe each of these categories in detail, and suggest methods by which the required changes may be made.

Note that it is *not* mandatory for an application to migrate all its modules and resources to the 32-bit programming environment. OS/2 Version 2.0 allows mixed model programming, where 32-bit applications may make use of existing 16-bit modules and resources. The subject of mixed model programming is discussed in Chapter 13, "Mixing 16-Bit and 32-Bit Application Modules."

12.1 Data Types

A number of the function type declarations and data type definitions used by Presentation Manager map into different "standard" C language type definitions under OS/2 Version 2.0. This is due to the differences between the 16-bit architecture of previous versions and the 32-bit architecture of Version 2.0.

For example, the Presentation Manager data type EXPENTRY, used to declare exportable entry points, is defined under OS/2 Version 1.3 in the following way:

```
#define EXPENTRY     pascal far
```

However, the *pascal* linkage convention is not used in the 32-bit OS/2 Version 2.0 programming environment, since all function calls use the standard C calling convention. The only exception is when creating applications that access 16-bit functions; see Chapter 13, "Mixing 16-Bit and 32-Bit Application Modules."

In addition, *far* memory references are not used, since the 32-bit flat memory model allows addressability to all locations in the process address space. In the 32-bit programming environment therefore, the EXPENTRY type is defined as blanks; 32-bit applications are not required to use the EXPENTRY keyword.

The OS/2 header file *os2.h* provided with the *IBM Developer's Toolkit for OS/2 2.0* provides transparent remapping of the Presentation Manager type definitions to

their new C language equivalents, and thus no modification is required to existing functions and data definitions that use the Presentation Manager types. However, applications that have used standard C language type definitions in place of the Presentation Manager types will require modification. For this reason, it is recommended that *all* Presentation Manager applications should use the Presentation Manager function and data type definitions.

12.2 Function Name Changes

Under OS/2 Version 2.0, certain function names for operating system kernel functions and Presentation Manager functions have been changed to provide improved consistency. These changes obey the following rules:

- The use of the terms *Create*, *Get*, *Set* and *Query* in function names is in accordance with SAA conventions, as follows:
  - *Create* implies that a new resource is created as the result of a function call.
  - *Get* implies that a resource is made available to the application as the result of a function call.
  - *Set* implies that a system value is changed as a result of a function call.
  - *Query* implies that a system value or handle to an existing function is returned as the result of a function call.
- Verbs are placed before nouns in function calls.
- Similar actions have similar semantics, although they may operate on different types of resources.

These changes have been made in order to provide improved consistency in function names, simplifying the task of learning the various function names.

Applications that use operating system functions should be checked and function names altered where necessary. A list of corresponding function names in OS/2 Version 1.3 and OS/2 Version 2.0 is given in *OS/2 Version 2.0 - Volume 1: Control Program*.

12.3 32-Bit Interface Constraints

The following subsystems, present in OS/2 Version 1.3, have no 32-bit equivalents, since they are not portable and are device-dependent.

VIO Video subsystem; these function calls should be replaced with Presentation Manager GPI calls or AVIO (advanced video) calls.

KBD Keyboard subsystem; these function calls should be replaced by processing WM_CHAR messages from a Presentation Manager application.

MOU Mouse subsystem; these function calls should be replaced by processing WM_MOUSEMOVE and WM_BUTTON messages.

MON Device monitor subsystem; these function calls should be replaced by using appropriate message queue hooks in a Presentation Manager application.

Note that these subsystems are supported under OS/2 Version 2.0 as 16-bit service layers, for access by existing applications. However, OS/2 Version 2.0

does not provide thunk layers to enable 32-bit applications to access these service layers. 32-bit applications that use these subsystems must employ mixed model programming techniques to access the 16-bit services; see Chapter 13, "Mixing 16-Bit and 32-Bit Application Modules" for further discussion.

Only two forms of input/output arc supported in the 32-bit environment; the *stdin/stdout* C interface (producing command line-based applications that can be run in a text window) and Presentation Manager applications. Programs wishing to perform graphics, handle keyboard or mouse input, or intercept and modify device information should do so using Presentation Manager functions.

Certain other functions have also been removed from the 32-bit programming interface. These are primarily concerned with the management of memory in the segmented memory model. Therefore functions such as **DosSizeSeg()**, **DosLockSeg()** and **DosUnlockSeg()**, **DosR2StackRealloc()** etc are not implemented in the 32-bit programming interface, although the 16-bit entry points are still supported for compatibility purposes.

12.4 Function Enhancements

A number of enhancements have been made to existing operating system functions in OS/2 Version 2.0. These include the semaphore functions for synchronization between threads and processes, and the **DosCreateThread()** function for initiating secondary threads.

12.4.1 Semaphore Functions

The enhanced semaphore functions available under OS/2 Version 2.0 were described in Chapter 2, "Operating System/2." Previous versions of OS/2 provided only basic semaphore facilities with limited capability to handle multiple events. Where these semaphores were used in applications under previous versions of OS/2, they may be updated to the new OS/2 Version 2.0 semaphore facilities using the following guidelines:

- Where a semaphore is used to serialize the access to a particular data object or system resource from multiple threads, a *mutex* semaphore should be used.
- Where a semaphore is used to signal an event occurring in one thread to other threads having an interest in this event, an *event* semaphore should be used.
- Where an application waits upon the clearing of one semaphore from a range of semaphores using the **DosMuxWaitSem()** function, this function may be replaced by the use of a *muxwait* semaphore.

  The muxwait semaphore has additional flexibility over the use of the **DosMuxWaitSem()** function with normal semaphores under previous versions of OS/2, since with a muxwait semaphore, a thread may wait for any *one* of a list of mutex semaphores or event semaphores to be cleared (as with previous versions), or may wait for *all* of the semaphores to be cleared. This latter capability is not available under OS/2 Version 1.3.

The enhanced semaphore functions available under OS/2 Version 2.0 are described in the *IBM OS/2 Version 2.0 Control Program Reference*.

12.4.2 Thread Management

The **DosCreateThread()** function has been enhanced in OS/2 Version 2.0 to facilitate the creation of secondary threads. Under previous versions of OS/2, an application was required to explicitly allocate a memory segment for the stack to be used by a secondary thread. Under OS/2 Version 2.0 however, stack allocation is performed as a built-in part of the **DosCreateThread()** function, and deallocation is performed automatically by the operating system upon termination of a thread.

Applications that use the **DosCreateThread()** function should be modified to use the new form of the function call, as shown in Figure 123.

```
APIRET    rc;                          /* Return code              */
PTID      ThreadID;                    /* Thread identifier        */
MYSTRUCT  *ParmBlock;                  /* Initialization data      */

rc = DosCreateThread(ThreadID,         /* Create thread            */
                     Thread,           /* Entry point for thread   */
                     ParmBlock,        /* Parameters for thread    */
                     0L,               /* Start immediately        */
                     8192);            /* Stack size for thread    */
```

Figure 123. DosCreateThread() Function. This example shows the enhanced form of this function as implemented under OS/2 Version 2.0.

Note that the **DosCreateThread()** function under OS/2 Version 2.0 also allows parameters to be passed to the thread as part of the **DosCreateThread()** function. The third parameter to the function is a 32-bit pointer, which may be used to pass the address of an application-defined data structure containing the required parameter data.

The **DosCreateThread()** function and its implementation under OS/2 Version 2.0 are described in greater detail in *IBM OS/2 Version 2.0 Control Program Reference*.

OS/2 Version 2.0 also allows an application to forcibly terminate a thread using the **DosKillThread()** function. This function allows an application's primary thread to terminate any secondary threads prior to its own shutdown, in a more elegant manner than was possible under previous versions of OS/2.

12.5 Memory Management

The 64KB segment size limitation imposed by the Intel 80286 processor architecture has been eliminated in OS/2 Version 2.0, avoiding the need for applications to allocate large data structures in individual units of 64KB. The flat memory model implemented under OS/2 Version 2.0 allows applications to request the allocation of individual memory objects up to 512MB in size, using the **DosAllocMem()** function, an example of which is shown in Figure 124 on page 263.

```
APIRET    rc;                          /* Return code                  */
PVOID     pObject;                     /* Pointer to memory object     */

rc = DosAllocMem(&pObject,             /* Allocate memory object       */
                 73727,                /* Size of memory object        */
                 PAG_COMMIT |          /* Commit memory immediately    */
                 PAG_READ   |          /* Allow read access            */
                 PAG_WRITE);           /* Allow write access           */
```

Figure 124. DosAllocMem() Function. This function replaces the DosAllocSeg() function implemented in previous versions of OS/2.

The above example shows the allocation of 73KB of memory as a single memory object, with read and write access permitted. The PAG_COMMIT flag is set in the **DosAllocMem()** function call, in order to commit the storage immediately.

12.6 New Presentation Manager Functions

A number of new functions have been added to Presentation Manager under OS/2 Version 2.0. While these functions do not add new capabilities to the Presentation Manager interface, they simplify application development by combining operations that previously required multiple steps into a single Presentation Manager function call.

Table 5 (Page 1 of 2). New Presentation Manager Functions in OS/2 Version 2.0

Function	Description
WinInsertLboxItem()	Inserts a listbox item
WinDeleteLboxItem()	Deletes a listbox item
WinSetLboxItemText()	Sets the text of a specified listbox item
WinQueryLboxCount()	Returns the number of items in a listbox
WinQueryLboxSelectedItem()	Returns the offset (item number) of the selected item in a listbox
WinQueryLboxItemText()	Returns the text of a specified listbox item
WinQueryLboxItemTextLength()	Returns the length of the text of a specified listbox item
WinPopupMenu()	Creates and presents a context (popup) menu
WinCheckMenuItem()	Sets a check mark against a pulldown menu item
WinIsMenuItemChecked()	Determines whether a menu item is currently checked
WinEnableMenuItem()	Enables or disables a menu bar or pulldown menu item
WinIsMenuItemEnabled()	Determines whether a menu bar or pulldown menu item is currently enabled
WinSetMenuItemText()	Sets the text of a specified menu bar or pulldown menu item
WinFileDlg()	Displays the standard SAA-conforming file dialog box
WinDefFileDlgProc()	Default processing function for subclassing file dialog box
WinFontDlg()	Displays the standard SAA-conforming font dialog box
WinDefFontDlgProc()	Default processing function for subclassing font dialog box

Table 5 (Page 2 of 2). New Presentation Manager Functions in OS/2 Version 2.0	
Function	**Description**
WinQueryButtonCheckstate()	Determines the current check state of a check box or 3-state button.
WinSetDesktopBkgnd()	Sets the current desktop background
WinQueryDesktopBackground()	Queries information about the current desktop background

A number of other functions are also added in OS/2 Version 2.0, and are used to deal with the passing of messages between 16-bit and 32-bit modules in Presentation Manager applications. These functions are discussed in Chapter 13, "Mixing 16-Bit and 32-Bit Application Modules."

These functions are described in more detail in the *IBM OS/2 Version 2.0 Presentation Manager Reference*.

12.7 Summary

Existing 16-bit applications written for OS/2 Version 1.3 may execute under OS/2 Version 2.0 without modification. However, significant enhancements to performance and functionality are possible by taking advantage of additional features provided by the 32-bit OS/2 Version 2.0 environment. In order to take full advantage of the 32-bit environment, applications must be modified to use the new features.

In addition, a number of changes have been made in the 32-bit programming environment to provide improved consistency and ease of use, simplifying the task of learning the operating system interfaces and reducing the amount of coding required by application developers. The incorporation of these changes into applications will also require source code modification.

It is not necessary for applications to migrate *all* their modules and resources to the 32-bit environment, since OS/2 Version 2.0 allows mixing of 16-bit and 32-bit code and resources within the same application. However, 32-bit modules that make calls to 16-bit modules or resources must be aware of the differences in addressing schemes between the the 16-bit and 32-bit environments.

Chapter 13. Mixing 16-Bit and 32-Bit Application Modules

Under OS/2 Version 2.0, 32-bit applications may make use of existing 16-bit application code and resources; this practice is known as **mixed model programming**. For example, a 32-bit Presentation Manager application may access an object window procedure contained in a 16-bit DLL created for OS/2 Version 1.3. This capability allows 32-bit applications to make use of existing application objects, avoiding the necessity to rewrite existing object libraries to accommodate the 32-bit programming environment.

Applications that make use of 16-bit modules and resources must be aware of the particular characteristics of the 16-bit environment, which affect the way that application modules interface with one another and pass parameters or messages. Such characteristics include:

- Pointers in the 16-bit environment are made up of a segment selector and an offset; this addressing scheme is therefore known as **16:16**. Pointers in the 32-bit environment are composed of a linear offset only; hence the addressing scheme is known as **0:32**. Note that this difference in representation applies not only to memory pointers, but also to window and resource handles under Presentation Manager.
- Memory objects passed as parameters between 16-bit and 32-bit routines must not be greater than 64KB in size, in order to avoid problems with the 16-bit segmented memory model.
- Memory objects passed as parameters from 32-bit applications to 16-bit routines must not lie across a segment boundary.

Obviously, conversion of pointers and possible realignment of memory objects is required when passing control between 16-bit and 32-bit modules. This conversion between addressing schemes is known under OS/2 Version 2.0 as **thunking**. Thunking is performed using a simple algorithm known as the **Compatibility Region Mapping Algorithm (CRMA)**. This algorithm, along with sample code, is described in *OS/2 Version 2.0 - Volume 1: Control Program*.

13.1 Function Calls to 16-Bit Modules

Thunking considerations affect the way in which a 16-bit function must be declared within the 32-bit module, and the way in which parameters to be passed to the 16-bit function are defined. Such functions and parameters must be declared using the *#pragma linkage* directive and the ***far16*** keyword, as shown in Figure 125.

```
#pragma stack16(8192)

USHORT MyFunction(USHORT FirstNum, HWND _Seg16 hWnd);

#pragma linkage (MyFunction, far16 pascal)
```

Figure 125. Declaring a 16-Bit Function in 32-Bit Code

Note the use of the ***#pragma stack16*** directive to set the stack size for all 16-bit function calls made from the 32-bit module.

Declaring a 16-bit function in this manner will cause the operating system to automatically perform thunking for all "value" parameters (that is, those other than pointers). Pointers passed as parameters must be explicitly defined using the ***_Seg16*** keyword, as shown in Figure 125.

The *#pragma linkage* and *#pragma stack16* directives are discussed in more detail in the *IBM C Set/2 User's Guide*.

13.2 Using 16-Bit Window Procedures

A 32-bit application may access window procedures that reside in 16-bit modules, either statically linked or as DLLs. However, the differences between addressing schemes require some consideration on the part of the developer, since both the window handles and any pointers passed as message parameters will differ in their representation.

13.2.1 Creating a Window

When a 32-bit application module creates a window, and the window procedure for that window resides in a 16-bit module (either statically linked or in a DLL), the calling routine must explicitly declare the 16-bit nature of the window procedure's entry point when registering the window class. This may become rather complex, since it involves invoking a 32-bit entry point from a 32-bit module, but passing a 16-bit entry point as a parameter.

A simpler solution is to build a registration routine within the 16-bit module, which registers the window class and creates the window. The 32-bit module then need only invoke this routine, and allow for the resulting 16-bit window handle. This technique has the added advantage that Presentation Manager records the fact that the window was registered from a 16-bit module, and will automatically perform thunking for system-defined message classes. The technique is illustrated in Figure 126 on page 267.

Since the 16-bit module would typically be a DLL, the registration routine is declared in the 16-bit module as an exportable entry point using the EXPENTRY keyword.

The 32-bit module declares the registration routine *MakeMyWindow()* as a 16-bit function using the *#pragma linkage* directive with the *far16* keyword. Since in 16-bit code, the EXPENTRY keyword forces use of the *pascal* calling convention, the directive also specifies this calling convention. Note that if the registration routine and the window procedure were to reside in a DLL, this declaration would typically take place within a header file provided by the developer of the DLL.

The 32-bit module invokes the registration routine that registers the window class and creates the window. The registration routine then returns the window handle to the 32-bit module, which stores it in 16:16 format. Note that the registration routine in the 16-bit module is not aware that it is being called from a 32-bit module.

```
32-bit Module

#pragma stack16(8192)

HWND MakeMyWindow(void);                    /* 16-bit function prototype  */
#pragma linkage (MakeMyWindow, far16 pascal)

HWND _Seg16 hWindow;                        /* 16:16 window handle        */
  :
  :
hWindow = MakeMyWindow();                   /* Call registration routine  */

16-bit Module

HWND EXPENTRY MakeMyWindow(void)            /* Registration routine       */
{
     HWND hCurrWindow;                      /* 16:16 window handle        */

     WinRegisterClass(...);                 /* Register window class      */
     hCurrWindow = WinCreateWindow(...);    /* Create window              */

     return(hCurrWindow);                   /* Return 16:16 window handle */
}
```

Figure 126. Creating a 16-bit Window From Within a 32-bit Module

This approach allows the same DLL to be accessed by both 16-bit and 32-bit applications concurrently. The developer of the DLL simply provides two separate header files containing declarations of the DLL's entry points, in the appropriate format for each programming environment.

13.2.2 Passing Messages to 16-Bit Windows

Passing data between 16-bit and 32-bit window procedures via message parameters also requires consideration of the internal representations of the data types passed within the parameter. For system-defined message classes, this is handled automatically by OS/2 Version 2.0, but for application-defined message classes the conversion between addressing schemes must be handled by the application, since the operating system has no way of determining the intended contents of each parameter.

Simple "value" parameters (such as integers, characters, etc.) may be passed without the need for translation. It is recommended that message parameters be constructed using the standard Presentation Manager macros described in 6.6.6, "Creating Message Parameters" on page 93.

When a pointer or handle is passed in a message parameter to a 16-bit window procedure, the pointer or handle must be translated to the 16:16 addressing scheme by the application. Since the 16-bit module is unlikely to have been written with code to achieve this conversion, it is the responsibility of the 32-bit module.

Conversion may be achieved using the *_Seg16* keyword to explicitly define a 16:16 pointer or handle, which is then placed in a message parameter using the MPFROMP macro. This is illustrated in Figure 127 on page 268.

```
typedef struct mystruct {                    /* Define data structure      */
                        CHAR * _Seg16 Name;
                        ULONG         ulA;
                        ULONG         ulB;
                        USHORT        usC;
                        } MYSTRUCT;

pragma seg16(MYSTRUCT)                       /* Define pragma directive    */

MYSTRUCT * _Seg16 MyStruct;                  /* 16:16 pointer              */

APIRET   rc;                                 /* Return code                */

MPARAM   mp1;                                /* Message parameter          */

rc = DosAllocMem(&MyStruct,                  /* Allocate data structure    */
                 4096,                       /* Size of data structure     */
                 PAG_READ  |                 /* Allow read access          */
                 PAG_WRITE |                 /* Allow write access         */
                 PAG_COMMIT);                /* Commit storage immediately */

<Initialize structure if required>

mp1 = MPFROMP(MyStruct);                     /* Set message parameter      */
```

Figure 127. Passing a 16:16 Pointer as a Message Parameter. This example shows the 32-bit code necessary to define and initialize a 16:16 pointer to be passed to a 16-bit window procedure.

The resulting message parameter may then be passed to a window in a 16-bit module using the normal **WinPostMsg()** or **WinSendMsg()** functions, using a 16:16 window handle obtained in the manner shown in Figure 126. Note that the data structure referenced by the pointer may not be greater than 64 KB in size, and must not cross a segment boundary. This is ensured in Figure 127 by using the ***#pragma seg16*** directive, since a structure defined using this pragma will automatically be aligned on a segment boundary by the C Set/2 compiler.

Note also that defining a structure with the *#pragma seg16* directive does not implicitly qualify pointers within the structure with the *_Seg16* keyword. Such pointers must be explicitly qualified, as shown in Figure 126. Further information on the *#pragma seg16* directive can be found in the *IBM C Set/2 User's Guide*.

A 0:32 pointer may also be converted to a 16:16 pointer using the **DosFlattoSel()** function provided by OS/2 Version 2.0. This function provides the correct remapping of pointer formats from the 32-bit flat memory model to the 16-bit segmented memory model.

13.2.3 Passing Messages to 32-Bit Windows

The technique described above handles messages passed to a window in a 16-bit module. However, messages passed from that window to the 32-bit module may also require thunking. In order to perform this thunking, the 32-bit application may define a **thunk procedure** and register this procedure to Presentation Manager, which then invokes the thunk procedure whenever a message is passed from within the window.

This registration is achieved using the **WinSetWindowThunkProc()** function, which is illustrated in Figure 128 on page 269.

```
WinSetWindowThunkProc(hWindow,                /* Window handle          */
                      (PFN)ThunkProc16to32); /* Thunk proc entry point  */
```

Figure 128. Mixed Model Programming - WinSetWindowThunkProc() Function

The **WinSetWindowThunkProc()** function call is made from the 32-bit module. Since the window class for the window has been registered in the 16-bit module, Presentation Manager recognizes that the thunk procedure is to handle 16-bit to 32-bit conversion.

A thunk procedure may be deregistered, by issuing a **WinSetWindowThunkProc()** function call with the thunk procedure entry point address set to NULL.

Whenever Presentation Manager invokes a thunk procedure for a message, it passes the normal four parameters accepted by a window procedure, along with the entry point address of the window procedure to which the message was to be passed. This may be the window procedure defined for the destination window when its class was registered, or a subclass window procedure defined by the application. Thus thunking may take place, irrespective of whether a window has been subclassed.

A sample thunk procedure is shown in Figure 129.

```
MRESULT EXPENTRY ThunkProc16to32(HWND hwnd,       /* Window handle       */
                                 ULONG ulMsg,     /* Message identifier  */
                                 MPARAM mp1,      /* Message parameters  */
                                 MPARAM mp2,
                                 PFNWP wpWindow); /* Window procedure    */
{
        switch (ulMsg)
                {
                case WMP_MSG1:
                     mp1=DosSeltoFlat(mp1);       /* Thunk parameters    */
                     mp2=DosSeltoFlat(mp2);
                     break;
                case WMP_MSG2:
                     mp1=DosSeltoFlat(mp1);       /* Thunk 1st parameter */
                     break;
                }
        return((*wpWindow)(hwnd,                  /* Call window proc    */
                           ulMsg,
                           mp1,
                           mp2));
}
```

Figure 129. Mixed Model Programming - Thunk Procedure

The thunk procedure is invoked whenever a message is passed by the window in the 16-bit module to a window in the 32-bit module. The thunk procedure is similar in structure to a "normal" window procedure, but need contain cases only for application-defined message classes, since thunking for system-defined message classes is performed by Presentation Manager.

Note that since the thunk procedure is invoked by Presentation Manager, it must use the *system* linkage convention, and is thus declared using the EXPENTRY keyword.

In Figure 129, the 16-bit window contains two application-defined message classes, WMP_MSG1 and WMP_MSG2. The first of these contains pointers in both parameters, and thus both parameters are thunked by the thunk procedure. The second message class contains a pointer in the first message paramater only; the second may contain an integer or some simple value parameter which does not require explicit thunking. Thunking is performed using the **DosSeltoFlat()** function provided by OS/2 Version 2.0.

After performing the necessary thunking, the thunk procedure directly calls the window procedure entry point supplied by Presentation Manager when the thunk procedure is invoked. Note that this is one of the few instances where direct invocation of a window procedure should be used. The correct sequence of message processing is preserved in this case because the thunk procedure itself is invoked either synchronously or asynchronously by Presentation Manager, depending upon whether the message was sent or posted by the 16-bit window.

An alternative to the use of the **DosSeltoFlat()** function is the explicit use of the Compability Region Mapping Algorithm discussed in *OS/2 Version 2.0 - Volume 1: Control Program*. This algorithm is implemented in the subroutine *CRMA16to32* shown in Figure 130.

```
PVOID CRMA16to32(PVOID pPointer)                    /* Perform conversion  */
{
      USHORT usTemp;                                /* Temporary variable  */

      if (pPointer)                                 /* If not NULL         */
         {
         usTemp=HIUSHORT(pPointer) >> 3;            /* Shift right 3 bits  */
         return(MAKEP(uTemp,                        /* Swap hi & lo words  */
                      LOUSHORT(pPointer)));
         }
      else
         return(NULL);
}
```

Figure 130. 16:16 to 0:32 Address Conversion

The use of the **DosSeltoFlat()** function should be the preferred option, since the CRMA routines may fail under certain circumstances. Explicit use of CRMA should be restricted to those situations where special processing must be performed on the parameters being passed.

13.3 Summary

OS/2 Version 2.0 allows applications and resources from both 16-bit and 32-bit environments to coexist and communicate. A 32-bit application may make function calls to 16-bit code, and 16-bit and 32-bit window procedures may pass messages between one another.

Conversion between the 16:16 and 0:32 addressing schemes is achieved using thunks, which implement an algorithm known as the Compatibility Region

Mapping Algorithm. The IBM C Set/2 compiler provides transparent thunking for most function calls and parameters, using the *#pragma linkage* directive in conjunction with the *_Seg16* keyword.

For threads that will make calls to 16-bit code, the stack must also be aligned on a 64KB segment boundary, to avoid possible problems with stack overflow in the 16-bit code. Again, the IBM C Set/2 compiler facilitates this alignment through use of the *#pragma stack16* directive, which causes the thread's stack to be automatically aligned on a 64KB boundary.

Thunking becomes slightly more complex when communicating between 16-bit and 32-bit window procedures, since pointers passed in message parameters must be thunked. While Presentation Manager provides transparent thunking for all system-defined message classes, application-defined messages must be thunked explicitly by the application.

Presentation Manager provides some assistance to the application developer by allowing thunk procedures to be registered for a window. Presentation Manager automatically invokes the thunk procedure whenever a message to passed to that window from a window of another memory model.

The ability to mix 16-bit and 32-bit code in the same application provides considerable flexibility and protects investment in existing application functions and resources. This in turn eases the task of migrating the organizational development environment from the 16-bit to the 32-bit environment, since the transition need not be accomplished in a single step.

Chapter 14. Compiling and Link Editing an Application

A Presentation Manager application, written in a high-level language such as C, is compiled and link-edited in a similar fashion to a regular OS/2 or DOS application. System Object Model object classes are created in a similar way. However, additional steps are required for the creation of a Workplace Shell object class, prior to compilation of the C source code.

This chapter describes the process of creating an executable Presentation Manager application or system object model object class. The chapter describes the process of precompilation for system object model object classes, and the compilation, link edit and resource compilation stages for both system object model object classes and Presentation Manager applications.

The following diagram shows the overall process used to create a new class in the Workplace Shell:

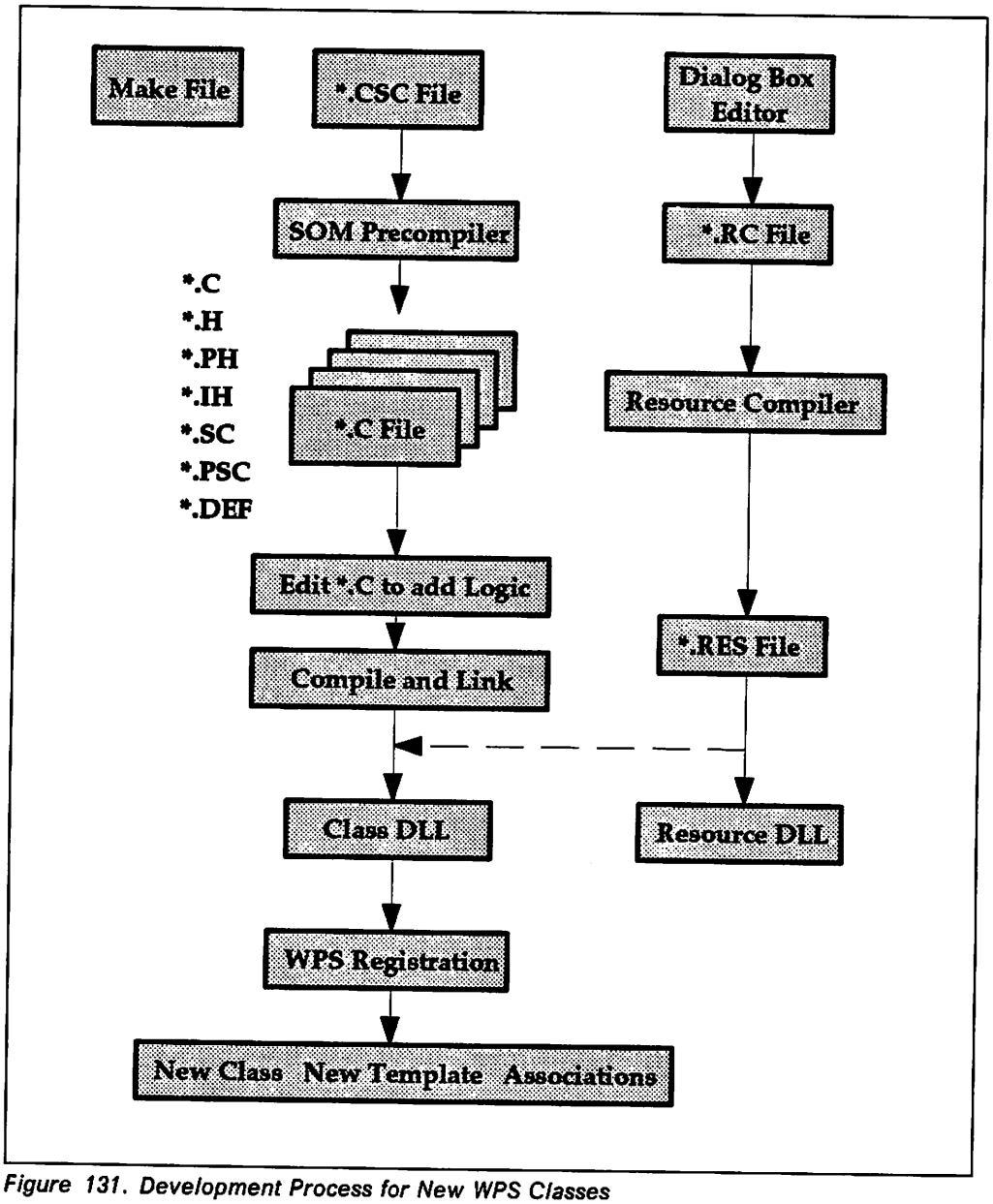

Figure 131. Development Process for New WPS Classes

The starting point for a system object model class is the class definition file which, for classes written using the C language, has an extension of .CSC. The class definition file is used as input to the SOM Precompiler, which will generate a number of files from the class definition file:

.H A public header file for programs that use the class.

.PH A private header file, which provides usage bindings to any private methods implemented by the class.

.IH An implementation header file, which provides macros, etc., to support the implementation of the class.

.C A template C file, to which code may be added to implement the class.

.SC A language-neutral class definition.

.PSC A private language-neutral core file, which contains private parts of the interface for the class.

.DEF An OS/2 DLL module definition file containing the relevant exports needed to implement the class.

14.1 Running the SOM Precompiler

Once the class definition file has been created, the SOM Precompiler is used to generate the source files for the class. The options for the SOM Precompiler are described in detail in the *IBM SOM Programming Reference*; however, a brief description of the options used to create the folder example is given below.

14.1.1 The Makefile

The instructions to be used are placed into a *makefile*. The new make facility *NMAKE*, provided with the IBM Developer's Toolkit for OS/2 2.0 is extremely flexible and rich in function. It is recommended that programmers read the NMAKE section of the *IBM OS/2 Version 2.0 Programming Tools Reference* if unfamiliar with makefiles.

The SOM Precompiler environment variables are set as follows:

```
!if [set SMINCLUDE=.;$(SCPATH);] || \
    [set SMTMP=$(SOMTEMP)] || \
    [set SMEMIT=ih;h;ph;psc;sc;c;def]
!endif
```

The use of the *!if* directive is somewhat confusing, as the statement has nothing to do with a conditional command. In fact, the OS/2 SET command is being executed to initialize the environment variables. The NMAKE utility executes any OS/2 command placed within square brackets in a *!if* directive.

The SMEMIT environment variable tells the SOM Precompiler which C source files are to be generated; the suffixes correspond to the file types described earlier in this chapter.

The SMINCLUDE and SMTMP environment variables are set from two NMAKE macros, which are defined at the top of the makefile:

```
SCPATH  = D:\toolkt20\sc
SOMTEMP = .\somtemp
```

The SMINCLUDE variable tells the SOM Precompiler where to find the class definition (.SC) include files.

The SMTMP variable locates the SOM Precompiler temporary workspace directory.

This brings us to the next part of the makefile that is responsible for ensuring the existence of the temporary directory.

```
!if [cd $(SOMTEMP)]
!  if [md $(SOMTEMP)]
!    error Error creating $(SOMTEMP) directory
!  endif
!else
!  if [cd ..]
!    error Error could not cd .. from $(SOMTEMP) directory
```

```
! endif
!endif
```

This code checks for the existence of the directory and if it cannot be found, attempts to create it.

14.1.2 SOM Precompiler Invocation

The SOM Precompiler is invoked as a consequence of the following NMAKE inference rule:

```
.csc.c:
    sc $<
```

This infers that to go from a .CSC file to a ".C" file the SOM Precompiler (sc.exe) will be invoked on the .CSC file ("$<").

Note that for the folder example, many default SOM Precompiler options are used. Readers may wish to investigate some of the other options that are available, by checking the *IBM SOM Programming Reference*.

14.2 Compiling C Source Code

The following files are required to generate a system object model object class or Presentation Manager application from "C" source code:

Source An application may contain one or more source modules; multiple source modules are normally bound at link-edit time. However, application routines may be compiled and placed in a DLL for binding at execution time.

H An application may use a header file that contains data type definitions, constants and macro definitions for use by the application.

DEF A **module definition file**, while not required for all Presentation Manager applications, is definitely recommended. This file contains information for input to the link editor, as described in 14.2.1, "Module Definition File" on page 278.

RC A resource script file contains definitions for Presentation Manager resources, and/or statements that include resources from other files.

DLG If an application uses a dialog box, a dialog file is normally created containing a definition for the dialog box and its control windows. This file is normally included into the resource script file using an appropriate statement.

The role of these files in the development of a Presentation Manager application is illustrated in Figure 132 on page 277.

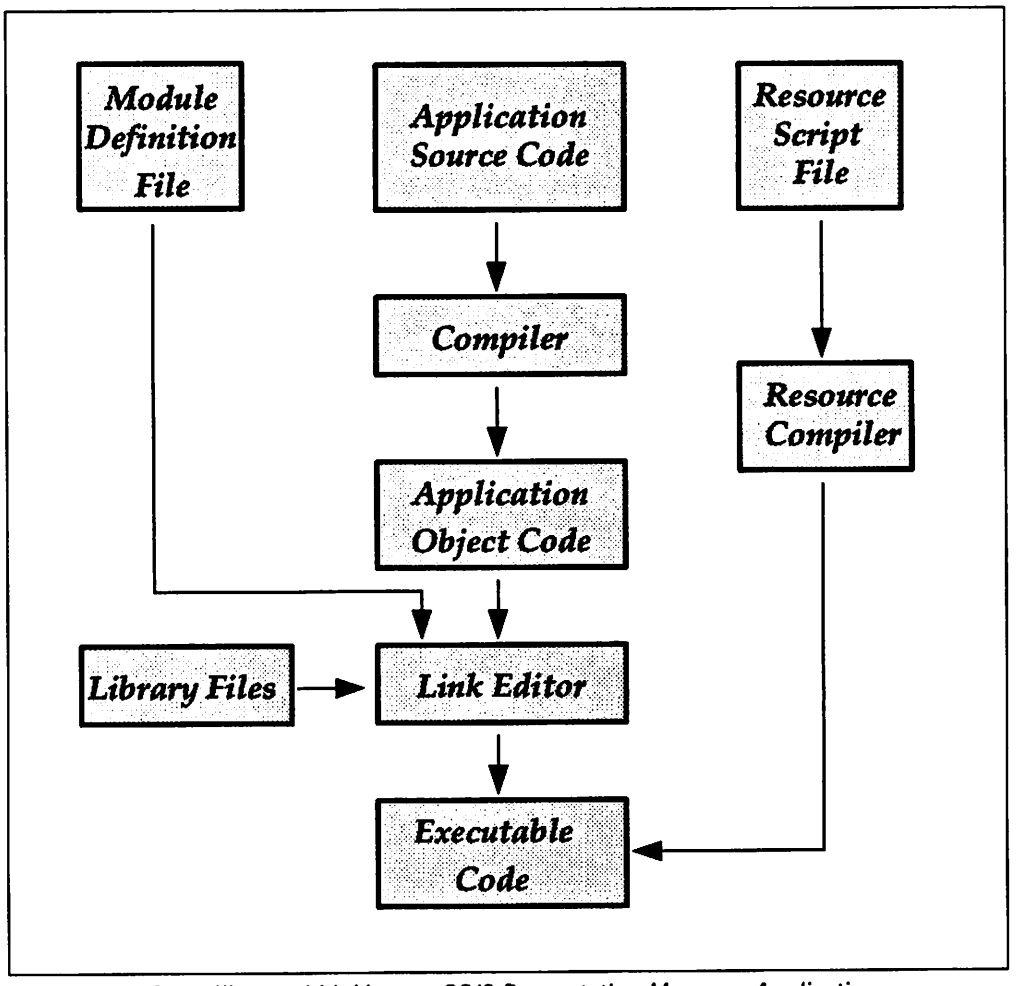

Figure 132. Compiling and Linking an OS/2 Presentation Manager Application

Note that certain additional header files are generated by the SOM Precompiler and used in compiling a system object model object class. These files are typically header files used to link the class with its parent, and have no direct bearing on the compilation process.

The following steps are required to compile and link-edit a Presentation Manager application:

1. Compile the source file using a high-level language compiler, to produce an object file (usually with an extension of .OBJ).
2. Link-edit the object file with any other required object files, and possibly one or more run-time libraries, to create an executable file (usually with an extension of .EXE).
3. Compile any Presentation Manager resources using the resource compiler, and incorporate these resources into the application's executable code or place them in a DLL module.

A number of compiler and link editor options are specified for Presentation Manager applications; these are shown in the example below. The executable file may then be run as a Presentation Manager application under OS/2.

Note that the third step above is not required if Presentation Manager resources are not used by the application. However, it is envisaged that virtually all Presentation Manager applications will make some use of resources, due to the

benefits to be gained through externalization of logic-independent or language-specific application components.

Note that the command sequences for compiling and link editing an application may be combined and placed in a parameterized OS/2 command file for automated execution. Alternatively, a **makefile** may be created and the NMAKE utility used to compile and link edit the application.

14.2.1 Module Definition File

The module definition file is a simple text file required by most Presentation Manager applications. Module definition files are used when link editing both programs and dynamic link libraries. By convention, the module definition file has the same name as the program or library source file, but with an extension of DEF.

A sample module definition file for a simple Presentation Manager application is shown Figure 133:

```
; Sample Presentation Manager Module Definition File

NAME            MYPROG   WINDOWAPI
DESCRIPTION     'Sample PM Application (C) IBM Corporation 1991'
PROTMODE

STUB            'OS2STUB.EXE'

STACKSIZE       8192
HEAPSIZE        1024

EXPORTS         ThisWindowProc
                ThatWindowProc
                TheOtherWindowProc
```

Figure 133. Sample Module Definition File for Presentation Manager

A module definition file normally begins with a **NAME** or **LIBRARY** statement, which identifies the module as a program or DLL respectively, and assigns it a module name. The keyword **WINDOWCOMPAT** may be used to specify a full-screen application that may be run in a Presentation Manager text window, or **WINDOWAPI** may be used to specify a full Presentation Manager application.

The module definition file also contains a **DESCRIPTION** entry, containing text that is embedded by the link editor into the header of the executable module. This text may contain information such as a copyright notice or author information concerning the module.

The **PROTMODE** keyword should be used to indicate that the application will be run only in protect mode under OS/2 (note that this is a standard provision; all Presentation Manager applications must run in protect mode). This allows the link editor to shorten the header in the executable module.

The **STUB** keyword instructs the link editor to set up a stub file that generates an error message if the user attempts to execute the application in a DOS environment. OS2STUB.EXE is a DOS executable file that performs this function, and is provided with the *IBM Developer's Toolkit for OS/2 2.0.*

A **DATA** statement may be used in the module definition file to indicate the disposition of data segments created by the module. Data segments may be specified as **SHARED**, **NONSHARED** or **NONE**. If SHARED is specified, different processes using the code segments of a dynamic link library will share the same data segments; if NONSHARED is specified, the operating system will create a new set of data segments for each process using a DLL. NONSHARED is recommended.

The **STACKSIZE** and **HEAPSIZE** statements specify the size of the memory areas to be reserved for the application's stack and for the local heap in the program's automatic data segment. Note that the recommended minimum stack size for Presentation Manager applications is 8 KB. Note also that STACKSIZE is not used for dynamic link library modules, since a DLL has no stack (see 14.5.1, "Creating a DLL" on page 281).

The module definition file may also contain an **EXPORTS** statement for all exportable entry points (such as window procedures) window procedures contained in the module. This statement causes the entry points for window procedures to be defined in the header of an executable module, so that they may later be called from outside the current executable module (since window procedures are actually invoked by Presentation Manager on behalf of the application, rather than directly by the application itself).

An **IMPORTS** statement may be used to define the entry points for those functions and/or resource definitions that will be imported from a dynamic link library. However, an IMPORTS statement is not required if an **import library** is being link edited with the application (see 14.5.2, "Using a DLL" on page 282).

The module definition file, and the statements it may contain, are described fully in the *IBM OS/2 Version 2.0 Application Design Guide*.

14.2.2 Compiler Options

Using the IBM C Set/2 compiler, the following command sequence is recommended:

```
cc /C+ /L+ /G3 /Ti+ MYPROG
```

The **/C+** option indicates that only the compile step should be run, and not the link-edit step, which is then performed explicitly at a later point using the LINK386 utility. The **/L+** option causes the compiler to produce a source listing file.

The **/G3** option optimizes the code for execution on an Intel 80386 processor. The code will also execute on an 80486 processor. However, for code that will mainly be executed on 80486 hardware, use of the **/G4** option is recommended. See the *IBM C Set/2 User's Guide* for further information.

The **/Ti+** option instructs the compiler to generate symbolic debugging information which may then be used when debugging the application with the IBM C Set/2 debugging tool.

14.3 Link Edit

When using the IBM C Set/2 compiler, an application may be compiled and link-edited in a single operation. The following command sequence is used to compile and link edit a program for Presentation Manager under OS/2 Version 2.0:

```
CC /L- /G3 /Ti+ /Gm+ MYPROG OS2386.LIB MYPROG.DEF
```

This command sequence directs the linkage editor to compile the file MYPROG.C to produce an object file named MYPROG.OBJ, and to link this file to produce an executable file named MYPROG.EXE, without creating a list file, to use the run-time library named OS2386.LIB and the module definition file named MYPROG.DEF.

Note the use of the **/Gm+** option. This option causes the multithreading "C" run-time libraries to be used in linking, thereby enabling multithreadingin the resulting application code.

14.4 Resource Compilation

Presentation Manager resources used by the application and defined in a resource script file are compiled using the resource compiler provided in the *IBM Developer's Toolkit for OS/2 2.0*. The command sequence used to invoke the resource compiler is as follows:

```
rc MYPROG.RC MYPROG.EXE
```

This command sequence causes the resource compiler to read the resource definitions from the resource script file MYPROG.RC and compile them to produce an intermediate resource file MYPROG.RES, which is then incorporated into the executable module MYPROG.EXE.

Resources may also be compiled and incorporated into dynamic link libraries, simply by specifying the name of the DLL rather than the EXE file when invoking the resource compiler. Note that certain additional considerations may apply; see 14.5.3, "Presentation Manager Resources in a DLL" on page 282 for further information.

Note that under OS/2 Version 2.0, 32-bit applications may use existing 16-bit application modules and resources. This concept is known as mixed model programming, and is discussed in Chapter 13, "Mixing 16-Bit and 32-Bit Application Modules."

14.5 Dynamic Link Libraries

Since window procedures that communicate using the standard Presentation Manager message conventions have no exterior interfaces aside from these messages, window procedures are essentially "black boxes" and are invisible to the application. It is thus possible to produce a number of window procedures to perform standard or often-used functions, and to store these window procedures in a library from which they may be accessed by a number of applications.

The recommended way to achieve this capability is to place such window procedures in an OS/2 dynamic link library.[5] Applications using these window procedures then reap the benefits of DLLs; namely that changes to window procedures contained in the DLLs do not require the applications to be re-linked and that multiple applications may use the same memory-resident copy of the window procedure code, since DLLs are re-entrant. The use of DLLs therefore maximizes the potential for code reuse, and facilitates the containment of change within a single application module.

Note that where existing 16-bit DLLs contain application objects, functions or resources required by 32-bit applications under OS/2 Version 2.0, these applications may access the 16-bit DLLs. This technique, known as mixed model programming, is discussed in Chapter 13, "Mixing 16-Bit and 32-Bit Application Modules."

14.5.1 Creating a DLL

To build a DLL from a particular source code module, the code is compiled in a similar manner to that used for any other Presentation Manager application code. However, use of the **/Ge-** option will cause the C Set/2 compiler to produce a DLL module.

```
CC /L- /G3 /Ti+ /Gm+ /Ge- MYDLL OS2386.LIB MYDLL.DEF
```

This command sequence instructs the compiler to take the file MYPROG.C, to compile and link edit it to produce a dynamic link library named MYPROG.DLL, and to use the file named MYPROG.DEF as input to the linking process. Use of the **/Gm+** option enables multithreading, and use of the **/Ge-** option directs C Set/2 to produce a DLL module.

A sample module definition file for use when creating a DLL is shown in Figure 134:

```
; Sample PM Module Definition File for Creating a DLL

LIBRARY          MYDLL INITINSTANCE TERMINSTANCE
DESCRIPTION      'Sample PM Dynamic Link Library (c) IBM 1991'
PROTMODE
DATA             MULTIPLE

EXPORTS          RoutineNumberOne
                 RoutineNumberTwo
                 RoutineNumberThree
```

Figure 134. Sample Module Definition File to Create a DLL

The following rules apply to module definition files when used to create dynamic link libraries:

- The **LIBRARY** statement must be used, and the module name declared must be the same as the file name of the DLL. The keywords **INITINSTANCE** and **TERMINSTANCE** may be used to indicate that any initialization code should be executed for each process which accesses the DLL.

[5] See 2.7, "Dynamic Linking" on page 20 for an explanation of the concept of dynamic linking.

- If the DLL will be accessed by separate processes within the system, separate data segments should be created for each process; this is achieved by specifying **DATA MULTIPLE** in the module definition file
- The **STACKSIZE** statement must not be used since a DLL does not use a stack.
- An **EXPORTS** statement must be included, defining each functicn that will be exported from the DLL.

The DLL module must be copied to a directory referenced by the LIBPATH statement in CONFIG.SYS. This is typically the C:\OS2\DLL directory, although another directory may be used if so desired.

14.5.2 Using a DLL

When an application is to use functions contained in a DLL, these functions must first be declared at compilation time. This is typically achieved by the use of an include file containing function prototypes for all exportable entry points within the DLL; such an include file is normally created along with the DLL and supplied to all application developers who will use the DLL.

Once the functions are declared within the application's source code, the external references must be resolved at the time the program is link edited. This may be performed in one of two ways:

- The functions may be identified in the application's module definition file using an **IMPORTS** statement, as described in 14.2.1, "Module Definition File" on page 278. For example, the statement:

  ```
  IMPORTS MYPROG.func1
  ```

  defines the function *func1*, which will be imported from a DLL named MYPROG.DLL.
- The functions may be included in an **import library**, which is a library specified at link edit time. An import library is created using the IMPLIB utility provided with the IBM Developer's Toolkit for OS/2 2.0. For example, the command:

  ```
  implib MYLIB.LIB MYLIB.DEF
  ```

  causes IMPLIB to create an import library named MYLIB.LIB, using information contained in a module definition file named MYLIB.DEF. The import library file should be placed in a directory referenced by the INCLUDE environment variable.

Note that the file OS2.LIB, used when link editing OS/2 application programs, is in fact an import library containing definitions for OS/2 system functions that are themselves implemented using DLLs.

14.5.3 Presentation Manager Resources in a DLL

As noted in 2.7, "Dynamic Linking" on page 20, Presentation Manager resources may be defined and stored in a DLL for use by multiple applications. However, the implementation of dynamic linking under OS/2 requires that each DLL must contain an executable module, and resources are not executable modules in their own right. If a DLL will contain only resources and will not contain an executable routine such as a dialog procedure, a dummy executable module must be provided for the DLL by the application developer, as shown below:

```
int ARCTUSED = 1;

void EXPENTRY dummy()
{
}
```

The declaration for the variable *ARCTUSED* is provided in order to avoid an error generated by the IBM C/2 compiler when it cannot find a module named *main* within the source file.

Editor's Note

The above statement may not be true for the C Set/2 compiler. This must be determined.

This dummy module is compiled and link-edited into an executable file in the normal way. The resource compiler is then used to compile and incorporate the resource definitions into the executable file.

14.5.4 Using Dialogs in System Object Model Objects

In the folder example, a dialog box is used to prompt the user for the folder's password, and for the user to enter the password. Creating and invoking the dialog is done in the normal way. However, invoking an object's methods from within a dialog procedure requires that the dialog procedure know the pointer to the object that invokes the method (that is, *somSelf*). This is done through the use of the *pCreateParams* parameter of the **WinDialogBox()** function. In this way, the pointer to somSelf is passed to the dialog procedure as follows:

```
pCreateParams = (PVOID)somSelf;
```

The dialog procedure may then store the pointer in the reserved window word QWL_USER:

```
case WM_INITDLG:
     WinSetWindowULong(hwndDlg,      /* Set window word          */
                       QWL_USER,     /* Offset of window word    */
                       (ULONG) mp2); /* Value to be stored       */
     break;
```

When an instance method must be invoked from the dialog procedure, the object pointer may easily be retrieved from the window words and used to invoke the method.

```
{
   PWFolder *aPWF;                          /* Object pointer      */

   PWF_INFO pwfolderInfo;                   /* Folder info struct  */

   aPWF = (PWFolder *)WinQueryWindowULong( /* Get object pointer  */
                      hwndDlg,              /* Dialog box handle   */
                      QWL_USER);            /* Offset of win word  */

   _QueryInfo(                              /* Invoke method       */
              aPWF,                         /* Object pointer      */
              &pwfolderInfo);               /* Folder info struct  */
}
```

In the above example, a **WinQueryWindowULong()** call is used to retrieve the object pointer from the window word, and store it in *aPWF*. This variable is then used as the first parameter when invoking the method *_QueryInfo*.

Note that the method name *_QueryInfo* is in fact fully defined as *pwfolder_QueryInfo*. However, as noted in 7.3.3, "C Implementation of an Object Class" on page 119, the SOM Precompiler automatically generates a macro to define the abbreviated form of the function name, in order to avoid the necessity for the programmer to type the full name.

14.6 Summary

The steps required in compiling and link-editing a Presentation Manager application are generally similar to those required to generate any other OS/2 application. Some files are required in addition to those used by a "conventional" application, due to Presentation Manager's use of externalized definitions for application resources such as menus, string tables and dialog boxes.

The module definition file provides a mechanism whereby various attributes of an application or dynamic link library may be specified. The module definition file also allows the developer to specify copyright information that is imbedded in the executable code.

The creation of dynamic link libraries, while essentially similar to that of normal application code, requires certain special considerations, notably with regard to the module definition file and the options that are specified at link-edit time. Code modules and Presentation Manager resources that are likely to be subject to change, or are of a potentially reusable nature may be placed in dynamic link libraries and thus isolated from the remainder of the application. The remainder of the application code is thus protected from changes that may be necessary to these dynamic link libraries in order to accommodate changes in such areas as organizational procedures or government legislation. Dynamic link libraries may also be used by multiple applications concurrently, by virtue of their reentrant nature.

Chapter 15. Adding Online Help and Documentation

In line with the philosophy of making applications easy to use through the provision of an intuitive, graphical user interface, it is extremely useful to have an application provide online, context-sensitive help and tutorial facilities to the end user. Such facilities further encourage learning by exploration.

Presentation Manager provides such capabilities in the form of the **Information Presentation Facility (IPF)**, which allows **help panels** to be generated and displayed in **help windows** under Presentation Manager. These help windows are linked with the normal application windows in such a way that when the user presses the "Help" key, selects a "Help" item from the menu bar or presses a "Help" button in a dialog box, the appropriate help window is displayed.

In addition to providing online help for Presentation Manager applications, IPF can also be used in a stand-alone mode, to provide online documentation for applications, or for business procedures that are independent of any single application. Using capabilities provided by IPF, an online procedure manual may initiate appropriate applications, thus providing a "real-life" tutorial capability for new or inexperienced users.

This chapter will describe the use of IPF for creating help libraries and online documents, and examine some ways in which IPF may improve the ease-of-use of Presentation Manager applications.

15.1 Creating Help Information

Help information is created in ASCII source files using an **IPF tag language**, which embeds formatting tags in the text. These tags define the formatting characteristics of the text which appears in help windows, and the appearance of the windows themselves. Once the source files are created, the **IPF compiler** is then used to translate the source files into an IPF library format. The IPF compiler can generate a table of contents and an index for help information.

15.1.1 IPF Tag Language

The IPF tag language is similar in structure and syntax to the Generalized Markup Language (GML) used by IBM's Document Composition Facility* (DCF*) product. Source files are simple ASCII text files, and can be created using any normal text editor.

Tags are embedded in the files simply by inserting the tag into the text at the required point. For example, this document was created and formatted using GML tags, and the opening sentences of this chapter were created using the format shown in Figure 135 on page 286.

```
:h1.Adding Online Help and Documentation
:p.In line with the philosophy of making applications easy to use
through the provision of an intuitive, graphical user interface,
it is extremely useful to have an application provide online,
context-sensitive help and tutorial facilities to the end user.
Such...
```

Figure 135. IPF Tag Language Example

Note that the chapter heading is preceded by a "header level 1" tag which causes the header text to be formatted in a particular manner. Similarly, the header is followed by a "paragraph" tag which caused a new paragraph to begin. The formatting of the text within the source file is not significant; there is no requirement for a tag to begin at the left margin, or for each tag to begin on a separate line. A developer may organize the source files in the most appropriate manner for readability.

The first statement in a source file must be a *:userdoc* tag, and the last statement must be a *:euserdoc* tag. These tags are required by IPF. Comments may also be imbedded within a source file using the *:** tag; comments are ignored by the IPF compiler, and do not appear in the formatted text.

A complete description of the formatting tags available under the IPF tag language is beyond the scope of this document. Some examples are given in the remainder of this chapter, and each tag is described in detail in the *IBM OS/2 Version 2.0 Information Presentation Reference.*

15.1.2 Defining Help Panels

The IPF tag language contains "header" tags with levels from 1 to 6; by default, levels 1 to 3 define the start of a new help panel, and are automatically included in the table of contents for online documentation. Each help panel therefore begins with one of these header tags, followed by the help text to be displayed within that panel.

```
:h2 res=12345 x=left y=bottom cx=50% cy=25%.Help Window Heading
:p.The sequence of operations you have performed was never
envisaged by the person who wrote this program, and no help
information has been written into these panels to deal with
this contingency. You are therefore totally beyond help.
Exciting, isn't it?
```

Figure 136. Simple Help Panel Source

The *res=* attribute specified in the *:h1* through *:h6* tags is used by IPF to uniquely identify the help panel, and to enable an application to link to that panel when the user requests help. See 15.3, "Linking Help Windows With Applications" on page 291 for further information.

The *x=*, *y=*, *cx=* and *cy=* attributes define the position and size of the window within its parent. In the case of a help window, the parent is the application window from which the help window was invoked.

The header levels 1 to 3 are used when organizing help panels within the table of contents. Panels defined with header level 3 are grouped under the last

defined header level 2, and those defined with header level 2 are similarly grouped under the last defined header level 1. Thus the order of definition within the source file *is* significant.

Note that multiple help panels may be placed within a single source file, simply by including additional header tags. By default, header levels 4 to 6 will not cause the display of a new help window, but will appear as formatted subject headings within a help panel.

15.1.3 Displaying Graphics

In addition to text information, graphics may also be displayed in help panels. Either character graphics or bitmaps may be used. Character graphics are embedded directly within the help text, using the *:xmp* tag to ensure correct formatting. Bitmaps are referenced from within the help text using the *:artwork* tag. An example of this tag is shown in Figure 137.

```
:h2 res=223.Bitmap Help Example
:p.This example shows how to display a bitmap in a help window
using the :artwork tag.
:artwork name='BITMAP.BMP' align=left.
```

Figure 137. Displaying a Bitmap in a Help Window

The *align=* attribute on the *:artwork* tag allows a bitmap to be aligned either left-justified, right-justified or centered in the window. In situations where the bitmap must fill the entire window, the *fit* attribute may be specified, which causes the bitmap to be scaled to fit the window size.

15.1.4 Hypertext and Hypergraphics

Help windows may be nested through the use of links embedded within the help information. Both text and graphics may be defined as selectable; selection of these items can then be used to trigger events such as:

- Display of another help window with supporting help text
- Dispatch of message to the application window which invoked the help window
- Initiation of a new application.

Selectable items are known as *links*; text items are **hypertext links**, and graphical items are **hypergraphic links**. Each of these link types is described in the following sections.

15.1.4.1 Hypertext Links

Text items are defined using the *:link* tag, which defines the type of event triggered by the link operation, and details of the target object activated by the link. An example of a *:link* tag is shown in Figure 138.

```
:h2 res=004.Hypertext Example
:p.This example shows the use of a
:link reftype=hd res=1013.hypertext link:elink.
to display another help window when the user selects the
hypertext item.
```

Figure 138. Hypertext Link

The *:link* tag shown in the example above uses the *reftype=* attribute to identify the type of event to be triggered (in this case, the display of another help window). The *res=* attribute identifies the help window to be displayed when the hypertext item is selected.

Different values for the reftype= attribute will trigger different types of events:

- Specifying *reftype=hd* causes the display of another help window, as shown in Figure 138. The *res=* attribute must also be specified in order to identify the help window to be displayed.
- Specifying *reftype=fn* causes the display of a popup window containing a footnote. The *refid=* attribute must also be specified in order to identify the footnote.
- Specifying *reftype=inform* causes a message to be sent to the application window which invoked the help window. The *res=* attribute must also be specified, and its value is passed back to the application.
- Specifying *reftype=launch* causes another application to be started by the operating system. The name of the program to be started is specified in the *object=* attribute, and parameters may be passed to the program in the *data=* attribute.

The use of hypertext links provides a powerful means to develop sophisticated help text and online documentation, and to implement tutorials and "self-teaching" applications. This subject is discussed further in 15.8, "Self-Teaching Applications" on page 298.

15.1.4.2 Hypergraphic Links

Bitmapped graphic items are defined in a slightly different manner to text items, using the *:artlink* tag. The *:artlink* tag is specified on the line immediately following the *:artwork* tag in the source file.

Multiple *:artlink* tags may be defined for the same bitmap, in order that the user may select different portions of the bitmap to trigger different events. If multiple *:artlink* tags are specified however, they must be placed in a link file, which is referenced using the *linkfile=* attribute to the *:artwork* tag.

Both techniques are illustrated in Figure 139.

```
:h2 res=0005.Hypergraphic Example
:p.This example shows how to define hypergraphic links in a
bitmap.
:p.The first item shows a single hypergraphic link.
:artwork name='BITMAP.BMP' align=left.
:artlink.
:link reftype=hd res=0107.
:eartlink.
:p.The next item shows multiple hypergraphic links in the same
bitmap.
:artwork name='BITMAP2.BMP' align=center linkfile='BMP2'.
```

Figure 139. Hypergraphic Link

The use of a link file is mandatory where multiple hypergraphic links exist for the same bitmap. The format of a link file is shown in Figure 140 on page 289.

```
:artlink.
:link reftype=hd res=0110 x=0 y=0 cx=30 cy=20.
:link reftype=hd res=0111 x=31 y=21 cx=30 cy=20.
:link reftype=launch object='C:\APPLS\APPL1.EXE'
      x=61 y=41 cx=20 cy=10.
:eartlink.
```

Figure 140. Link File With Multiple Hypergraphic Links

Multiple *:link* tags are nested within a single *:artlink* tag, and define various selectable areas of the bitmap, each of which triggers a specific event when selected by the user. Any of the types of events normally triggered by a *:link* tag may be initiated from a hypergraphic link.

15.1.5 Viewports

Information within help windows is displayed in **viewports**. In the default case, a single viewport is defined that occupies the entire help window and contains the text. This is known as a **simple viewport**. However, multiple viewports may be defined within the same help window, and handled separately. For example, two viewports may be defined in a help window; the first may be used to display a graphical diagram, while the second may contain a text narrative relating to the diagram. The user may scroll the text in the window, but the diagram remains displayed since it lies in a separate viewport. This type of definition is known as a **complex viewport**.

Multiple viewports are normally defined within a help window using *:link* tags which are known as **automatic links**. These operate in a similar manner to hypertext links, but are invoked automatically when the help window is displayed. An example of a help window containing such links is shown in Figure 141.

```
:h2 res=0120
      x=center y=center width=50% height=50%.
      Multiple Viewports Example
:link reftype=hd res=0121
      auto dependent
      vpx=left vpy=bottom vpcx=50% vpcy=100%
      scroll=none titlebar=none rules=none.
:link reftype=hd res=0122
      auto dependent
      vpx=right vpy=bottom vpcx=50% vpcy=100%
      scroll=vertical titlebar=none rules=none.
```

Figure 141. Multiple Viewports Using Automatic Links

The *auto* attribute specifies that the viewport is to be opened automatically when its parent help window is opened. The *dependent* attribute specifies that the viewport is to be closed when its parent is closed. The *vpx=*, *vpy=*, *vpcx=* and *vpcy=* attributes specify the position and size of the viewport within the parent window. In Figure 141, two viewports are opened, positioned side-by-side within the parent window.

The *scroll=*, *titlebar=* and *rules=* attributes determine whether each viewport possesses its own scroll bars, title bar and sizing borders. In Figure 141, neither viewport contains a title bar or sizing border (both make use of the parent's title

bar and border), but the right-hand viewport contains its own scroll bar. This allows the right-hand viewport to be scrolled while the left-hand viewport remains unchanged.

15.1.5.1 IPF-Controlled Viewports

By default, the presentation of information in viewports is under the control of IPF, using instructions defined in the source files. Such viewports are known as **IPF-controlled viewports**. When an IPF-controlled viewport is used, text is automatically formatted within the viewport by IPF, and presented in the help window.

15.1.5.2 Application-Controlled Viewports

A viewport may also be defined as an **application-controlled viewport**, using the *:acviewport* tag. This tag allows an application to take direct control of a viewport, and to present information in this viewport in a manner determined by that application. For example, a full-motion video application could be used to display information in video format.

The *:acviewport* tag is shown in Figure 142.

```
:h1 res=0101
      x=center y=center width=50% height=50%.
      scroll=none.Application-Controlled Viewport Example
:acviewport dll='SAMPLES' objectname='flight' objectid=1
      vpx=left vpy=bottom vpcx=50% vpcy=50%.
```

Figure 142. Application-Controlled Viewport

The *:acviewport* tag causes IPF to load a dynamic link library as specified in the *dll=* attribute, and to pass control to the entry point identified by the *objectname=* attribute.

15.2 Compiling Source Files

Once the text has been generated in source files, it must be compiled in order to produce a **help library**. By default, source files have an extension of IPF. The IPF compiler does not require this extension, but if the compiler finds two files with the same name, the file with the IPF extension is used in the compilation.

For source files that will be used to produce **online documents** rather than help libraries, the extension INF should be used. See 15.6, "Stand-Alone Online Documentation" on page 297 for further information.

15.2.1 The IPFC Command

The IPF compiler is invoked using the IPFC command, as follows:

```
IPFC SOURCE.IPF /X /W3 >ERRORS.TXT
```

The above command invokes the IPF compiler with the input file SOURCE.IPF. The /X parameter instructs the compiler to produce a cross-reference listing for all headings, diagrams, etc. The /W*n* command specifies the level of error-reporting to be performed; valid levels are 1 (/W1) to 3 (/W3). The final parameter pipes any error messages to the file ERRORS.TXT in order that they may be examined later. The IPFC command is fully documented in the *IBM OS/2 Version 2.0 Information Presentation Reference.*

When creating online documentation that will function in a stand-alone format rather than as help associated with an application, the /INF parameter is specified. This causes the IPF compiler to search for source files with the INF extension, and to format the output for use with the online viewing utility VIEW.EXE.

15.2.2 National Language Support

Support for languages other than U.S. English may be provided in help files by specifying the /COUNTRY, /CODEPAGE and /LANGUAGE parameters in the IPFC command. These parameters affect the collating sequence used when creating a table of contents or index, and the titles displayed for note (*:nt*), warning (*:warning*) and caution (*:caution*) tags.

These parameters and the use of national languages in help windows and online documentation is described in more detail in the *IBM OS/2 Version 2.0 Information Presentation Reference*.

15.3 Linking Help Windows With Applications

In order for an application to display help information using IPF, a number of steps must be performed to link the application's windows with the corresponding help windows. Each of these steps is described in the following sections.

15.3.1 Creating a Help Table

A help table is a Presentation Manager resource, and is defined in a resource script file using the HELPTABLE resource. An example of a help table is shown in Figure 143.

```
HELPTABLE MAINHELP
BEGIN
     HELPITEM MAIN,    SUBTABLE_MAIN,    EXTHELP_MAIN
     HELPITEM DIALOG1, SUBTABLE_DIALOG1, EXTHELP_DIALOG1
END

HELPSUBTABLE SUBTABLE_MAIN
BEGIN
     HELPSUBITEM MI_FILE, 0010
     HELPSUBITEM MI_EDIT, 0020
     HELPSUBITEM MI_VIEW, 0030
     HELPSUBITEM MI_EXIT, 0040
END

HELPSUBTABLE SUBTABLE_DIALOG1
BEGIN
     HELPSUBITEM EF_ITEM1, 0101
     HELPSUBITEM EF_ITEM2, 0102
     HELPSUBITEM CK_ITEM3, 0103
     HELPSUBITEM PB_ITEM4, 0104
     HELPSUBITEM PB_ITEM5, 0105
END
```

Figure 143. Help Table Resource Definition

The HELPITEM resources within the help table define each application window for which help is to be provided, and point to a HELPSUBTABLE resource. Each HELPITEM resource also defines the panel identifier of the optional extended help panel for that window.

A HELPSUBTABLE resource is defined for each window, and contains HELPSUBITEM resources that identify each item within the window for which help is to be provided, and the panel identifier of the help panel for that item.

For example, Figure 143 shows a main window with the window identifier MAIN, and a dialog box with the identifier DIALOG1. A subtable is defined for MAIN, which defines the menu bar items within that window, and identifies a help panel for each of these items. For DIALOG1, the subtable specifies the identifiers of the control windows within the dialog box, and identifies a help panel for each control window. The identifiers of the help panels must correspond to the identifiers specified in the *res=* attribute of the header tags.

15.3.2 Creating a Help Instance

Once the help table for an application has been created, the application must pass this help table to IPF and create a **help instance**, using the **WinCreateHelpInstance()** function. This function is shown in Figure 144.

```
PHELPINIT HelpInit;
HWND      hHelp;

HelpInit=DosAllocMem(HelpInit,                  /* Allocate memory object  */
                     sizeof(HELPINIT),          /* Size of HELPINIT struct */
                     PAG_READ   |               /* Allow read access       */
                     PAG_WRITE  |               /* Allow write access      */
                     PAG_COMMIT;                /* Commit storage now      */

HelpInit->cb=sizeof(HELPINIT);                  /* Specify size of struct  */
HelpInit->pszTutorialName=NULL;                 /* No tutorial             */
HelpInit->phtHelpTable=MAINHELP;                /* Help table identifier   */
HelpInit->phtHelpTableModule=NULL;              /* Help table in EXE file  */
HelpInit->hmodAccelActionBarModule=NULL;        /* Resource in EXE file    */
HelpInit->idAccelTable=0;                       /* Resource in EXE file    */
HelpInit->idActionBar=0;                        /* Default used            */
HelpInit->pszHelpWindowTitle="Help";            /* Help window title       */
HelpInit->usShowPanelID=CMIC_HIDE_PANEL_ID;     /* Do not show panel ids   */
HelpInit->pszHelpLibraryName="APPLHELP";        /* Name of help library    */

hHelp = WinCreateHelpInstance(hAB,              /* Create help instance    */
                              HelpInit);        /* HELPINIT structure      */
```

Figure 144. WinCreateHelpInstance() Function

The **WinCreateHelpInstance()** function is normally called from an application's main routine, immediately after creating the application's main window, but before entering the message processing loop.

The **WinCreateHelpInstance()** function creates the application's main help window, which is initially invisible, and passes appropriate information to that window to enable the specified help library to be loaded and access to be obtained to required resources. See 15.5, "Main Help Window" on page 294.

15.3.3 Associating a Help Instance

Once the help instance has been created, it must be associated with an application window. The help instance is normally associated with the application's main frame window, and help may therefore be provided for any children of the frame window, including the menu bar and client window. The **WinAssociateHelpInstance()** function is used to associate a help instance with an application window; an example of this function is shown in Figure 145.

```
rc = WinAssociateHelpInstance(hHelp,          /* Help instance handle    */
                              hFrame);        /* Frame window handle     */
```

Figure 145. WinAssociateHelpInstance() Function

The **WinAssociateHelpInstance()** function is normally called from the application's main routine, immediately following the **WinCreateHelpInstance()** function call.

15.3.4 Ending a Help Instance

Upon termination of the application, the help instance should be ended using a **WinDestroyHelpInstance()** function call, as shown in Figure 146.

```
rc = WinDestroyHelpInstance(hHelp);           /* Destroy help instance   */
```

Figure 146. WinDestroyHelpInstance() Function

This function is invoked immediately after termination of the application's message processing routine, and prior to destroying the application's main window.

15.4 Displaying Help Panels

Help panels are displayed in help windows by IPF as a result of user interaction. A user may cause a help panel to be displayed in one of three ways:

- Hitting the F1 key
- Selecting a "Help" item on the menu bar or a pulldown menu
- Pressing a Help pushbutton in a dialog box.

Each of these actions normally results in a WM_HELP message being generated. This message is trapped by IPF, which then determines the active application window and uses the current help table to identify the help panel for that window.

15.4.1 F1 Key

Under the default accelerator table maintained by Presentation Manager for all windows, the F1 key causes a WM_HELP message to be generated and posted to the queue for the window that possessed the input focus when the key was pressed. Explicit definition of the accelerator key by the application is not required.

15.4.2 Help Menu Bar Item

A "Help" menu bar item, whether it is defined on the menu bar or in a "Help" pulldown menu, should be defined using the MIS_HELP style. This will cause the item to generate a WM_HELP message, rather than a WM_COMMAND message. The definition of such an item is shown in Figure 92 on page 199.

15.4.3 Help Pushbutton

A "Help" pushbutton in a dialog box should be defined using the BS_HELP and BS_NOPOINTERFOCUS styles. The WM_HELP message is passed to IPF, which then determines the control window within the dialog box that currently possesses the input focus, and displays the help panel for that control window.

15.5 Main Help Window

The **WinCreateHelpInstance()** function returns a window handle. This is the handle of the application's main help window. The main help window is created by IPF, with a standard format and with a window procedure supplied by Presentation Manager. Whenever a help panel is displayed, it appears as a child window of the main help window.

The main help window has a title bar, which contains a title specified by the application. The application passes this title in the pszHelpWindowTitle parameter to the **WinCreateHelpInstance()** function.

The main help window contains a menu bar with several pulldown menus allowing the user to perform text searches, view the index, etc. An application developer may modify this menu bar using the resource definitions contained in the *hmtailor.rc* file provided with IPF. This file includes an *hmtailor.h* file, which contains the integer constant definitions for the menu bar and pulldown menu items. Additional items may be defined within the help pulldown menu, but their resource identifiers should be between 7F00 and 7FFF to avoid conflicts with identifiers already defined.

When the menu bar of the main help window has been modified, it must be resource-compiled in the normal way, and combined with the application's executable file or with a DLL. Its resource identifier must then be specified in the *idActionBar* parameter of the **WinCreateHelpInstance()** function call. If additional accelerator keys have been defined, the identifier of the accelerator table must also be defined in the *idAccelTable* parameter.

If the menu bar and accelerator table definitions have been combined with a DLL, the module handle of this DLL must be specified in the *hmodAccelIActionBarModule* parameter. The module handle must first be obtained using the **DosLoadModule()** or **DosGetModuleHandle()** functions. These functions are described in 9.3.2, "Loading Resources From a DLL" on page 200.

15.5.1 The Help Pulldown Menu

The menu bar of the main help window contains a "Help" pulldown menu which in turn contains a number of options. The resource definition for this pulldown menu is included in the *hmtailor.rc* file provided with IPF, and illustrated in Figure 147 on page 295.

```
   :
   :
SUBMENU     "~Help",              ID_HELP
BEGIN
   MENUITEM "~Help for help...", ID_HELP_FOR_HELP
   MENUITEM "~Extended help...", SC_HELPEXTENDED, MIS_SYSCOMMAND
   MENUITEM "~Keys help...",     SC_HELPKEYS, MIS_SYSCOMMAND
   MENUITEM "Help ~index...",    SC_HELPINDEX, MIS_SYSCOMMAND
END
   :
   :
```

Figure 147. Help Pulldown Menu Definition. This example shows the "Help" pulldown menu included in the menu bar of the default main help window used by IPF.

When the user selects an item from the help pulldown menu, a message is generated, and is trapped by IPF and processed. Typically, this message causes IPF to query the application for the correct help panel. The message generated is dependent upon the pulldown menu item selected by the user.

15.5.1.1 Help For Help

When the user selects this item, IPF sends a WM_COMMAND message to the active application window, with the first parameter containing the ID_HELP_FOR_HELP identifier. The application's window procedure should process this message by sending an HM_REPLACE_HELP_FOR_HELP message containing the panel identifier of its own "Help for help" panel if one exists, or by sending an HM_DISPLAY_PANEL message with both parameters set to zero in order to display the default panel.

15.5.1.2 Extended Help

When the user selects this item, IPF responds by displaying the "Extended help" panel defined for the active application window in the help table. If no such help panel is defined for that window, IPF sends an HM_EXT_HELP_UNDEFINED message to the application (see 15.5.2.3, "HM_EXT_HELP_UNDEFINED" on page 296).

15.5.1.3 Keys Help

When the user selects this item, IPF sends an HM_QUERY_KEYS_HELP message to the active application window. The application's window procedure should process this message by returning the panel identifier of its own "Keys help" panel in the return code to Presentation Manager.

15.5.1.4 Help Index

Selecting this item causes IPF to display the index of the current help library.

15.5.2 Communication Between IPF and Applications

Information may be communicated from IPF to an application in response to user interaction in a help window. This information may be in the form of application events or errors, and is communicated to the application in the form of a message passed to the application window specified in the **WinCreateHelpInstance()** function. Such messages originate from the main help window, as part of the window procedure supplied by Presentation Manager for that window.

15.5.2.1 HM_INFORM Message

This message is passed to the application when the user selects a hypertext or hypergraphic item in a help window, for which the *reftype=inform* attribute has been specified. The first parameter of the HM_INFORM message contains the identifier specified by the *res=* attribute in the hypertext or hypergraphic link definition.

An application window typically processes the HM_INFORM message by examining the identifier in the first message parameter, and dispatching a message of the appropriate class to itself or another application window, in order to initiate the action requested by the HM_INFORM message.

15.5.2.2 HM_ERROR

This message is passed to the application when an error occurs during a user interaction with a help window. This message allows the application to display its own error message in such cases, thereby providing a consistent appearance for error messages. The first parameter of the message indicates the reason for the error. These reasons are documented in the *IBM OS/2 Version 2.0 Information Presentation Reference*.

An application typically processes the HM_ERROR message by displaying an appropriate message box and returning zero to Presentation Manager. If the application does not process the message, Presentation Manager takes no action.

15.5.2.3 HM_EXT_HELP_UNDEFINED

This message indicates that the user selected the "Extended help" item on the "Help" pulldown menu, and that no such help panel was defined for the active application window.

An application may process this message in one of three ways:

- Display a message box indicating that no help is available
- Display its own help window by explicitly creating and displaying the window on the screen
- Pass a HM_DISPLAY_HELP message back to IPF, instructing IPF to display a particular help panel.

If the application does not process this message, no panel is displayed and the user's request is simply ignored.

15.5.2.4 HM_SUBITEM_NOT_FOUND

This message indicates that the user issued a help request on an item for which no help panel is defined in the current help table. The application may process this message in one of three ways, as described in 15.5.2.3, "HM_EXT_HELP_UNDEFINED." The application should then return TRUE to Presentation Manager.

If the application does not process this message, the extended help panel for the currently active window is displayed by IPF.

15.6 Stand-Alone Online Documentation

Panels containing text produced using IPF need not be called from an application; IPF can be used to produce "stand-alone" online documentation, which is viewed using the VIEW.EXE utility provided with OS/2 Version 2.0. Online documents have similar capabilities to help libraries; both text and graphics may be included, and hypertext and hypergraphic links are supported.

Online documents also have a number of differences from help libraries:

- In an online document, a panel may be identified by using the *name=* or *id=* attributes in the heading, rather than the *res=* attribute. These attributes allow the use of alphanumeric characters, where the *res=* attribute must specify an integer identifier. Links are then defined using the *refid=* attribute in the *:link* tag.

  Note that the *name=* or *id=* attributes may not be used if files will be concatenated and hypertext or hypergraphic links are required between files.

- Online documents may not use hypertext or hypergraphic links with *reftype=inform*, since there is no associated application for the help instance, and IPF cannot determine the window to which the message should be directed.

- An online document has a main window created by IPF, which contains the table of contents for the document. The title of this main window is determined by a *:title* tag.

  Note that the :title tag may only be used for online documents, and not for help libraries. The title of an application's main help window is specified in the **WinCreateHelpInstance()** function call.

The ability to include hypertext and hypergraphics in online documents allows the creation of online procedure manuals that automatically invoke the appropriate application or applications for each step of the procedure. This is achieved by defining the hypertext or hypergraphic items with *reftype=launch*, specifying the name of the executable file for the required application. See 15.8, "Self-Teaching Applications" on page 298 for further discussion of such applications.

15.6.1 Compiling Online Documents

Source files that will be used for online documents are created in an identical manner to those used for help text. When the IPFC command is invoked to compile the source files, the /INF parameter should be specified, as follows:

```
IPFC SOURCE.IPF /X /W3 /INF >ERRORS.TXT
```

The IPF compiler will then include the necessary hooks to enable VIEW.EXE to display the online document. Note that online documents compiled with the /INF parameter have a default extension of .INF, rather than the normal extension of .HLP for help libraries.

15.6.2 Concatenating Source Files

Multiple document files may be concatenated to produce a single online document. Concatenation is achieved by creating an OS/2 environment variable that contains the names of the concatenated files. For example:

```
SET BIGDOC=DOC1.INF+DOC2.INF+DOC3.INF+DOC4.INF
```

This environment variable is typically set from within a batch file, which also contains the command VIEW BIGDOC to view the resulting concatenated document.

Hypertext links are permitted between panels in different files, but a panel must use the *res=* attribute in the heading tag to identify itself, rather than the *name=* or *id=* attributes. The *global* attribute must also be specified in the panel heading.

15.7 Application Tutorials

For any application, the developer or developers may supply a tutorial, which is effectively another Presentation Manager program which provides step-by-step guidance to the user. IPF enables tutorials to be started from within help windows, in a number of ways:

- Within a help panel, a hypertext or hypergraphic link may be defined with *reftype=launch*, and with the *objectname=* attribute specifying the name of the tutorial program. When the user selects the hypertext or hypergraphic item, the tutorial is started automatically.
- A "Tutorial" item may be included in the "Help" pulldown menu in the application's main help window. This is done automatically by IPF if any help panel heading tag (*:h1* through *:h6*) contains the *tutorial* attribute.

  In this case, the name of a tutorial program must be specified in the *pszTutorialName* parameter of the **WinCreateHelpInstance()** function call.

When the user selects the "Tutorial" item from the "Help" pulldown menu, an HM_TUTORIAL message is sent to the active application window, with the first parameter containing the name of the tutorial program. The application typically processes this message by calling the **DosExecPgm()** function to start the tutorial program.

15.8 Self-Teaching Applications

An extension of the tutorial concept is possible, where the user invokes an online procedure manual that describes a business process; hypertext and hypergraphic links can be imbedded in this manual to start the application or applications that support the business process. This can be performed in either of two ways:

- Where the steps in the business process are largely independent of one another, a separate application may be used for each step.
- Where the steps are interdependent, a single application can be used, with links triggering application events by way of messages.

Note that this is an extension of the **procedural entity** concept originally discussed in Chapter 3, "Object-Oriented Applications."

15.8.1 Loosely Coupled Applications

Where the steps in a business process are independent of one another and do not require any great coordination between supporting application functions, separate programs may be used to carry out each step. In this case, the procedure manual is created as an online document with links for each step. Each link is specified with *reftype=launch*.

15.8.2 Tightly Coupled Applications

Where interdependencies exist between the steps in a business process, and where the application functions that support these steps must therefore interact closely with one another, a single application is used. In this case, the procedure manual is created as a help library, and execution is handled as follows:

1. The application creates a help instance for the help library in the normal manner, creates its own main window but *does not* make this window visible.
2. The application's main window makes itself the active window, and sends an HM_DISPLAY_HELP message to the main help window to cause the initial help panel to be displayed.
3. Each step in the business is defined using hypertext or hypergraphic links with *reftype=inform*. When such an item is selected, it causes an HM_INFORM message to be posted to the application's main window (the active window).
4. When the application's main window receives the HM_INFORM message, it examines the message parameters to determine the required action, then creates one or more additional display windows or dialog boxes and makes these visible, allowing the user to complete the required step.
5. When the current step is complete, the user selects an appropriate menu bar item or pushbutton, and returns to the procedure manual.

Using this technique, the application's main window retains overall control of the application, and can ensure coordination between steps and impose a sequence of execution if this is required.

15.9 Summary

The Information Presentation Facility allows an application developer to create online context-sensitive help panels and stand-alone online documentation for Presentation Manager applications. The ability to link between panels and between a panel and applications provides a flexible and powerful tool for developing:

- Presentation Manager applications that contain comprehensive help information
- Online manuals both for applications and for business processes
- Interactive tutorials for applications and business processes
- "Self-teaching" business processes where the online manual automatically starts the required applications and leads the novice user through the process.

The flexibility of IPF and the high level of interaction between a help instance and its controlling application allows the application to exercise significant

control over the way in which help information is displayed to the end user. The ability to combine multiple viewports in a single help window allows the simultaneous use of text, graphics and other technologies such as image or full-motion video to provide help, documentation and tutorial information.

Chapter 16. Problem Determination

The steps required for identification and resolution of application errors and "bugs" in the Presentation Manager application environment are basically similar to those required for conventional programming environments. However, the event-driven nature of the Presentation Manager application model often causes unnecessary confusion when developers attempt to test and debug their applications. This chapter describes a simple approach to problem determination and resolution under Presentation Manager, which will help in locating and removing the majority of application problems.

Successful problem determination in the Presentation Manager environment, as in any programming environment, requires some basic ingredients:

- Effective problem documentation
- A methodical approach to problem resolution
- Knowledge and experience of the application environment
- A symbolic debugging tool such as CodeView or Multiscope**
- A measure of luck (!)

When these requirements are satisfied, problem determination may proceed through the following three phases:

1. Documentation
2. Isolation
3. Diagnosis and resolution.

The remainder of this chapter describes each of these phases in detail, discussing each step in the resolution process, and also describes the symptoms and likely solutions for some common application problems.

16.1 Problem Documentation

Problems in Presentation Manager applications typically occur within window procedures, or in subroutines invoked from within window procedures. This is not surprising, since all processing within a Presentation Manager application takes place as a result of messages, which are received and processed by window procedures. However, the event-driven nature of the Presentation Manager application model provides a built-in means of narrowing down the location of a problem, provided the event that caused the problem can be determined.

The initial documentation of an error may be performed by whoever is responsible for application testing, since no great level of technical expertise is required at this stage. It is important that the error is effectively documented in writing, at the time it occurs, along with relevant supporting information. Effective documentation greatly eases the task of recreating the error and identifying the underlying problem.

A worksheet that may be used for problem documentation, and that records the information required by the guidelines given in this chapter, is contained in Appendix D, "Problem Reporting Worksheet."

16.1.1 Window

In order to narrow down the location of the problem, it is first necessary to identify the window that was active when the error occurred. This is usually self-evident when the window is a display window, but may be less so if the window is an object window. However, an object window is activated upon receiving a message that typically originates from a display window, and therefore the problem may be effectively tracked down by beginning the search with the display window.

The first step is therefore to determine the display window with which the user was interacting when the error occurred. For documentation purposes, the window's title may be used to identify the window.

Step #1

Identify the window with which the user was interacting when the error occurred, and note its title.

Identification of the active window allows the search for the problem to be focused on the window procedure for that window. It is likely that the problem lies within that window procedure or a subroutine invoked from that window procedure. If not, the active window usually passes a message to another window which in turn causes the problem; this may be determined in the isolation phase (see 16.2, "Problem Isolation" on page 303).

16.1.2 Event/Action

Once the active window has been identified, it is necessary to determine the last user action before the error occurred. The name of the last menu bar or pulldown menu item, button or icon selected should be noted for documentation of the problem.

Step #2

Identify the last user action before the error occurred, and record the name of the menu bar or pulldown menu item, button or icon.

Identification of the last user action provides the initial location, within the window procedure for the active window, at which to begin searching for the problem. The problem is likely to be within the scope of processing for the message resulting from this action, or within that of another message generated during the processing of this action.

16.1.3 First Time vs Repetitive Actions

The third important step in documenting a problem is to determine whether the error occurs *every* time a particular action is performed, or if it only occurs after the action has been performed a number of times.

Step #3

Note whether the problem occurred when the action was performed for the first time, or only when the action had been repeated a number of times.

Problems that occur after a number of repetitions of an action typically indicate a resource limitation being exceeded, and provide a short-cut to problem resolution; see 16.3.2, "Repetitive Action Problems" on page 306.

16.2 Problem Isolation

Once the problem has been documented and narrowed down to a specific event within a particular window, the developer must determine the Presentation Manager message that results from that event, or the first such message if multiple messages are generated.

A symbolic debugging tool is then applied to the application code, and a breakpoint is set at the commencement of processing for that message class. The program is then single-stepped to determine the operation or function call at which the error occurs.

Step #4

Single-step with a symbolic debugging tool to determine the code statement at which the error occurs.

It is important during this stage to note any **WinPostMsg()**, **WinSendMsg()** or **WinBroadcastMsg()** function calls performed by the program, which will generate additional Presentation Manager messages in the system. If the initial pass through the processing for the current message does not reveal the error, the same process must be performed for each of these messages and the window procedures that process them.

Note that this single-stepping process is most useful in situations where the error occurs every time a particular action is performed. In cases where the error only appears after a large number of repetitions, single-stepping will be time-consuming and unproductive. In such cases, the problem resolution process may be expedited by omitting the isolation phase and immediately checking the logic of the processing for the failing message, to ensure that all resources allocated during processing are subsequently released. See 16.3.2, "Repetitive Action Problems" on page 306 for more details.

16.3 Problem Diagnosis

Once the cause of the error is narrowed down to a single statement within the source code, the problem with that statement must then be identified. Syntactical errors can generally be ruled out as a cause of failure during run time, since such errors are almost always identified during compilation of the application code. However, the items listed below are common causes of run-time errors, and should be checked for failing program statements:

- Logic: is the sequence of operations performed during the processing of a message in accordance with the application design? Have any steps been accidentally omitted?
- Parameters: are the correct variable names being used for parameters in the program statement? Do the parameter definitions in the program statement directly match those given in the function declaration?

- Pointers: do they contain valid references, and/or have they been correctly initialized prior to the program statement?
- Operating system or Presentation Manager resources: have they been successfully allocated prior to the program statement? Are resources released at the completion of processing for the event and if not, is there a valid reason for retaining them?

Step #5

Diagnose the cause of the problem by carefully checking the program statement, and correct the error.

The following sections provide descriptions of some common application problems against which failing programs may be checked.

16.3.1 First Time Problems

Problems that occur whenever a program statement is executed indicate an error in that statement or in a parameter used by that statement. These errors may often be indicated by the nature of the error. Some common errors are given below.

16.3.1.1 Trap 000D

This error indicates that the program attempted to access a location in memory that was not within the area allocated to its parent process. Since such access might violate the integrity of other applications or of the operating system itself, OS/2 disallows the access. Note that the pointer may directly reference a memory location, or may be the handle to a resource such as a window, presentation space etc.

The usual cause of such an error is that a pointer passed as a parameter in a function call is incorrect. The pointer may not have been initialized, or may have been set to an incorrect value as a result of a failed allocation request or incorrect pointer arithmetic.

Resolution actions are typically as follows:

- Check that the function call which allocated the resource referenced by the pointer completed without error, and that a valid pointer was returned.
- Ensure that any pointer arithmetic carried out on the pointer between allocation and the failing program statement is error-free.
- If the pointer is stored in an instance data area (that is, a data block normally stored in the window words), ensure that the pointer to the instance data area itself has been correctly read from the window words at the start of processing for the *current* message.

The allocation of a Presentation Manager resource may also fail for reasons associated with its parent window. See 16.3.1.3, "Failure to Allocate Resources" on page 305.

One additional cause of this error is the application releasing an instance data block too early in the processing of a WM_DESTROY message. If the memory object containing this data block is released, and the application then attempts to release other resources whose handles are contained within the data block,

OS/2 will not allow access to the memory. This problem is easily resolved by releasing the instance data block after other resources.

16.3.1.2 Trap 000E

This error indicates that an application under OS/2 Version 2.0 attempted to access an area in memory for which an address range had been allocated, but no storage committed. This error typically occurs when writing data objects into application data areas, since most operating system and Presentation Manager resources are automatically committed upon allocation.

The usual cause of such an error is that the application failed to include the PAG_COMMIT flag in the **DosAllocMem()** function call that allocated the resource, or failed to issue a **DosSetMem()** call when increasing the size of a memory object. The problem may be easily resolved simply by including the PAG_COMMIT flag or including a **DosSetMem()** call to ensure that sufficient storage is available before writing to a memory object.

16.3.1.3 Failure to Allocate Resources

A common error shows itself when Presentation Manager resources cannot be allocated correctly by an application. This occurs most frequently with resources allocated upon creation of a window, during processing of the WM_CREATE message.

The cause of the error is a failure, on the part of the application, to complete the default processing of the WM_CREATE message, *before* carrying out application-specific processing. Part of this default processing involves the allocation of a Presentation Manager control block for the window, allocation of a window handle etc. If this processing is not performed, via a **WinDefWindowProc()** function call, at the commencement of processing for the WM_CREATE message, function calls which use parameters such as the window handle will fail.

The problem may be easily resolved by placing a **WinDefWindowProc()** function call as the first statement in the processing for the WM_CREATE message.

16.3.1.4 Stack Space Exceeded

This error may appear in either of two places:

- If it appears during a call to an application subroutine, it usually indicates that the space reserved for the application's stack is insufficient for the number of nested function calls, local variables, etc., being used by the application.
- If it appears during a call to an operating system or Presentation Manager function, it may indicate the same cause as above, or that the limit of the application's Ring 2 stack, used by system-level code invoked by the application, has been exceeded.

The application's stack size may be exceeded in situations where the application makes a large number of nested subroutine calls, particularly where extensive recursion is used, and/or where large numbers of local variables are defined. In such cases, the stack may be need to be increased beyond the recommended minimum of 8KB, using the STACKSIZE statement in application's module definition file.

The Ring 2 stack limit is normally exceeded only in situations where a system-level function attempts to retrieve more items from the stack than were

originally placed there. This can occur where an application passes an incorrect parameter to a function; for example, if a parameter is declared by the function as an array of eight elements, but the application passes an array containing only seven elements, an error may occur when the function attempts to retrieve eight elements from the Ring 2 stack for processing.

While such an error may reveal itself during compilation, certain C typecasting conventions may mask the problem until run time. Where this error occurs during execution, careful checking of parameters is recommended.

16.3.1.5 Window Fails to Appear

This error occurs when an application issues a **WinCreateWindow()** or **WinCreateStdWindow()** function call to create a display window, but the window fails to appear on the desktop, even though the function returns a valid window handle. This error may result from either of two causes:

- The WS_VISIBLE flag may not be set in the frame creation flags for the window.
- The application may include the FCF_ICON or FCF_ACCELTABLE frame creation flags, but no icon or accelerator table resources are defined with resource identifiers which match the window identifier.

In both cases, the **WinCreateWindow** or **WinCreateStdWindow()** function will return a valid window handle, since the window has been created.

In the first case, the problem may be rectified by including the WS_VISIBLE flag in the frame creation flags, or by using the **WinShowWindow()** function to explicitly make the window visible.

In the second case, resources should be defined in the application's resource script file to match the FCF_ICON and FCF_ACCELTABLE frame creation flags. These resources must have identifiers that match the window identifier given in the **WinCreateWindow()** or **WinCreateStdWindow()** calls, since Presentation Manager uses this identifier to load the resources.

16.3.2 Repetitive Action Problems

Application errors that only reveal themselves after an action has been performed many times typically result from the application exceeding an operating system or Presentation Manager resource constraint. Resources such as window handles, presentation spaces, memory objects and so on, have finite limits. If an application repeatedly requests allocation of such resources without releasing them, these limits may be exceeded, in which case the resource will not be allocated and the application may fail when attempting to use the resource.

Such problems may manifest themselves as Trap 000D errors that will result in application termination, or may simply corrupt execution of the application. The effect is dependent upon the (invalid) contents of the resource handle when the application issues a function call that uses the resources. In certain cases, a function call may cause the application to enter an endless loop within the processing of one message, in which case the entire Presentation Manager desktop may "lock up."

This problem may be avoided by ensuring that all resource requests (**DosGet...()** and **WinGet...()** function calls) in the code are matched by corresponding

DosRelease...() and **WinRelease()** function calls. In accordance with the principle of encapsulating function, resources required for processing a particular message should be allocated, used and released during the processing of that message.

An exception to this rule occurs in the case of resources such as control blocks, presentation spaces for display windows, etc. These are typically allocated during processing of the WM_CREATE message, and persist throughout the life of the window, until released during processing of the WM_DESTROY message.

16.4 Post-Resolution Action

Once a problem has been identified and corrections made to the application, the resolution of the problem should be documented and placed, along with the original problem documentation, in some form of log. In this way, similar problems encountered at a later date may be more easily identified and resolved by reference to the log.

Step #6

After resolving the problem, document the resolution for future reference.

Logging systems for such information may range widely in complexity and sophistication, from simple paper files to automated database systems with keyword search capabilities. The level of system implemented by a development organization is dependent upon cost and the perceived productivity benefit to be gained from such information; organizations with an ongoing involvement in the development of complex Presentation Manager applications will derive greater benefits than those with only a single development project.

16.5 Summary

Problem determination in the Presentation Manager environment is similar to that for other application environments, and proceeds through a number of phases:

1. Documentation
   a. Failing window
   b. User action which caused the failure
   c. Whether the failure occurs upon first performing the action, or only upon repetitive actions
2. Isolation
3. Identification and resolution.

The documentation phase is normally part of the application testing cycle, and is performed by those responsible for such testing. No particular technical or programming skills are required for this phase.

Proper documentation of the failure usually allows a developer to determine the window procedure and message which caused the failure. This provides a useful starting point at which to search for the underlying problem.

The isolation phase is normally employed only for those failures that occur every time a particular action is performed, and involves the use of a symbolic debugging tool to single-step through the processing of the failing message, in order to determine the statement in the source code at which the error occurs.

Problems that occur only after many repetitions of a particular user action normally indicate that an operating system or Presentation Manager resource limit has been exceeded. This is usually the result of an application acquiring resources and failing to release them. In such cases, the resolution process may be expedited by immediately checking the processing of the offending message for resource allocation statements, and ensuring that each of these is matched by a corresponding statement that releases the resource.

Once the problem is narrowed down to a single application statement, the identification phase determines the cause of the problem and makes appropriate corrections to the source code. This phase requires familiarity with the OS/2 and Presentation Manager environments.

When the application has been corrected and submitted once more for testing, the problem and its resolution should be documented and this information made available for future problem determination activities. The availability of such information may be used to more quickly determine likely causes of similar problems in the future.

Chapter 17. Generic Application Modules

As mentioned throughout this document, the Presentation Manager application model promotes the reuse of application objects, by facilitating code modularity through data abstraction and encapsulation. With correct design procedures, it is possible to create generic application objects that may be used by multiple applications. Should subsequent applications require modification to allow different processing of particular message classes, this may be achieved through subclassing. The Workplace Shell application model introduced in OS/2 Version 2.0 provides even more potential for reuse, due to its enhanced support of inheritance and subsequently enhanced provision for object reuse.

Standardization and reuse of application code promotes consistency between applications in terms of processing techniques and user interfaces, and helps to enforce organizational programming and interface design standards. It also reduces the amount of new code required for applications, potentially shortening development time, and the use of previously developed and tested code may also decrease application testing time.

In situations where reusability at the application object level is either not possible or impractical, common application functions may still be developed as subroutines, and placed in libraries for access by multiple applications. The use of such subroutines reduces application development time.

This chapter examines the creation of such generic objects and subroutines within the Presentation Manager application model, and their placement in dynamic link libraries for subsequent use by applications. Since Workplace Shell objects are by definition placed in DLLs and available for reuse, they are not explicitly discussed in this chapter.

17.1 Generic Application Objects

Within the Presentation Manager application model, a window procedure provides both the definition of a data object and the methods that operate upon that data object. A window procedure may therefore be considered as a complete application object in its own right.

Where the data object to be manipulated by a window will be accessed by a number of applications, it makes sense to define the data object and its methods once, in a single window procedure, and to make that window procedure available to any application that needs to manipulate the data object. Any changes to the data object's characteristics or processing requirements can then be contained within the application object, avoiding the need to modify and recompile multiple applications. The dynamic linking capabilities of OS/2 facilitate such a technique, enabling such modifications to be automatically incorporated into applications at load time or run time.

17.1.1 Display Windows

A display window may be created as a generic application object, and its window procedure placed in a dynamic link library. The following steps are typically followed in creating such a window procedure:

1. Both the window procedure and a calling routine to create the window are written in the normal manner.
2. The routine containing the code to create the window is declared as an exportable entry point, and may thus be called by applications.
3. This routine returns the handle of the newly created window to the calling application, along with a success or failure code.
4. The source code is compiled and link edited as described in 14.5.1, "Creating a DLL" on page 281.

The calling application then simply issues a single function call, such as:

```
usSuccess = CreateEditWindow(hEdit);
```

The *CreateEditWindow()* function within the DLL handles all necessary operations including registration of the window class and creation of the window, places the resulting window handle in the address indicated by the *hEdit* parameter, and returns a success or failure code *(usSuccess)* to the calling application.

Note, however, that the above example assumes that the window is created with a predetermined title, size and position on the desktop. Should this not be the case, additional parameters to the *CreateWindow()* function would be required.

The definition of the *CreateEditWindow()* function as the only entry point in the DLL enforces the consistency of using this function. The calling application is still provided with the window handle, which allows it to communicate with the window and to subclass the window if required.

17.1.2 Object Windows

Object windows may be created and placed into dynamic link libraries in a similar manner to that already explained for display windows. However, object windows have an additional complication in that they are frequently created in secondary threads in order to handle long-running application tasks.

The steps in creating an object window for inclusion in a DLL are therefore as follows:

1. The window procedure, the calling routine to create the window and a routine to start the secondary thread from which the window is created, are written in the normal manner, as described in 10.1.1, "Threads Containing Object Windows" on page 206.
2. The routine containing the code to start the secondary thread is declared as an exportable entry point, and may thus be called by applications.
3. The source code is compiled and link edited as described in 14.5.1, "Creating a DLL" on page 281.

Note that the routine called by the application does not return the handle of the newly created window. Indeed, it cannot do so, since the creation of the window takes place asynchronously, in a secondary thread.

This obstacle is overcome by having the calling application (typically a window procedure) pass its own window handle as a parameter. This is passed to the

object window, which then passes an acknowledgement message to the calling window procedure, containing its window handle. The calling window procedure may subsequently communicate with or subclass the object window as required. This technique is described, along with an example, in 10.1.1, "Threads Containing Object Windows" on page 206.

17.1.3 Subclassing

When a generic application object (that is, window class) does not quite meet the requirements of an application, a developer may choose to use the generic object and modify its behavior, through subclassing, to meet the specific requirements. This may be easily achieved in conjunction with the methods described above, since the handle of the newly created window is either returned directly by the called routine or indirectly by the window itself. This handle can then be used in the **WinSubclassWindow()** function call.

The subject of subclassing is described in detail in 6.5.5, "Subclassing a Window" on page 84, along with examples of both the **WinSubclassWindow()** function and a subclass window procedure.

17.2 Dialog Boxes

Standard dialog boxes to handle commonly performed user dialogs may also be generated and placed in dynamic link libraries. The inclusion of a dialog box in a DLL however, is slightly more complicated than the inclusion of a "normal" display or object window, due to the definition of the dialog template as a Presentation Manager resource. The DLL must therefore include not only the dialog procedure and the invoking routine, but also the dialog template definition.

This necessitates the invoking routine within the DLL not only executing the **WinDlgBox()** function call, but also obtaining a module handle for the DLL and an entry point address for the dialog procedure. The necessary steps are as follows:

1. The dialog procedure and the invoking routine are developed and placed in a dynamic link library.
2. The invoking routine is declared as an exportable entry point and may thus be called by applications.
3. The dialog template is created using the Dialog Box Editor, resource compiled and combined with the DLL.

The invoking routine for a dialog box loaded from a DLL is described, along with an example, in Figure 94 on page 202.

17.3 Generic Subroutines

In addition to code reuse at the application object level, significant productivity gains can be achieved by the reuse of application code at the subroutine level, to carry out common operating system and Presentation Manager functions.

For example, initialization functions are required to perform tasks related to initializing the Presentation Manager environment and any data areas to be used by utility routines. Functions required to be performed include:

- Registration of the application to Presentation Manager
- Creation of a message queue
- Creation of an entry in the Workplace Shell Window List.

These functions may be combined into one or more standard initialization routines, which may be invoked upon entry to an application or thread. Examples of the necessary code are given in Figure 18 on page 76, Figure 19 on page 77 and Figure 20 on page 78.

Termination functions required by Presentation Manager applications could also be standardized; these functions include:

- Removing the application from the Workplace Shell Window List
- Destroying the application's main window
- Destroying the primary thread's message queue
- Deregistration of the application from the Presentation Manager environment.

These operations are performed by Presentation Manager upon termination of the application if the application does not perform them explicitly. However, it is recommended that the application carries out these actions, since they may then be achieved in a controlled manner. Examples of the necessary code are given in Figure 19 on page 77.

Other functions may often be required during the execution of an application, such as:

- Obtaining the window handle of the application's main client window. An example of this procedure is given in 6.6.5, "Identifying the Destination Window" on page 91.
- Passing a message from a subclass window procedure to the original window procedure for that window class. An example is given in Figure 25 on page 86.

These functions may also be combined into standard subroutines and placed in a library, thereby avoiding the need for application developers to repetitively code such functions.

A final function often used in the stepwise development of object-oriented Presentation Manager applications is a small routine to display a message box with the message "Action Not Yet Implemented," invoked when the user selects a menu bar or pulldown entry for which a method has not yet been coded. This function is typically invoked as the default case for the WM_COMMAND message class. In this way, methods within a window procedure may be implemented in a stepwise manner, and testing of existing methods may occur while new ones are being added. Selecting an action for which no method has yet been implemented will always result in the same message box being displayed.

17.4 Granularity

When placing generic code in a dynamic link library, whether that code is at the application object level or at the functional level, the question arises as to the way in which the code should be partitioned into individual DLL modules, and thus the level of granularity that will be achieved within the reusable code.

This decision must be made on the basis of interdependence; where routines are interdependent and are required or likely to be used together, it is advisable to

place them in a single dynamic link library. For example, a group of standard window manipulation routines would typically reside in single DLL.

However, generic application objects should bear no predefined relationship to one another, and generic window classes may therefore be used independently. In such a case, the window procedure and invoking routine for each window class should be placed in a separate DLL, along with any subroutines specific to that window procedure. Applications that desire to use more than one such application object may then access multiple DLLs.

17.5 Packaging

When a set of application objects and/or subroutines has been created and placed into a dynamic link library, the following items will have been generated:

- The dynamic link library containing the application objects and/or subroutines
- A header file containing declarations for the routines that will be called by applications in order to create the application objects or invoke the subroutines
- An import library file, containing entry point definitions for those routines that will be called by applications.

These items must be stored in a location from which application developers may access them. The use of a local area network to provide and manage access to such items is discussed in Chapter 18, "Managing Development."

In addition, appropriate entries must be included in an **interface control document**, which defines all common application objects and subroutines along with their external interfaces, and acts as a reference for application developers who wish to use such objects or routines.

17.6 Summary

The Presentation Manager application model affords the opportunity for significant standardization and reuse of application code, at both the application object and function levels. The dynamic linking facilities provided by OS/2 allow this capability to be carried over to executable code as well as source code. Such reusability reduces the amount of new code required for applications, thereby reducing the development time and cost of new applications.

Common application elements such as windows and dialogs may be defined and stored in dynamic link libraries for access by one or more applications, thus implementing reusability at the application object or dialog level. At a lower level, a large number of common Presentation Manager application tasks may be identified, which may also be placed in standard routines for purposes of enhancing programmer productivity.

A further benefit of using standardized routines is the improvement in the consistency of both the application code and the user interface. Such standardization provides an easy means of enforcing Systems Application Architecture CUA standards without the need for programmers to repetitively code definitions for an CUA-conforming user interface. In addition, the standardized implementation of various functions and techniques eases the task

of application maintenance, since all applications will behave in a similar manner through the use of common code.

Although the functions mentioned in this chapter are restricted to Presentation Manager functions, the same principles may be applied to other functions, dialogs etc., which are common to multiple applications within the organization. The creation of standard routines for such functions, and the incorporation of these routines into dynamically linked modules under OS/2, may enhance the modularity and granularity of applications and bring additional benefits through reduced development time for new applications, and through easier application maintenance and change management.

Chapter 18. Managing Development

In order to enable the implementation of large-scale applications in the programmable workstation environment, with multiple application developers participating in design, coding and testing, it is important not only to have an appropriate technological and architectural base for development, but also to provide appropriate and effective management of and control over the development process and development resources. Established techniques exist in the host-based application development environment for addressing such issues, but historically, the considerations of large-scale management and control have been overlooked in the workstation environment due to the relatively minor nature of workstation-based development projects in the past.

Two areas worthy of note in the workstation-based development environment are the management of developmental risk, and the management and control of development resources that are used and created during the application development process. This chapter will briefly describe these issues and offer some suggestions as to how they may be effectively resolved. Much of the discussion in this chapter will deal with Presentation Manager applications written using the C programming language, but the techniques described may be adapted to suit other environments and programming languages.

18.1 Risk Management

In any application development project, there are risks imposed by the use of new technologies and methods. These risks may be divided into two basic types:

- Technological risk; that which is imposed by new or unfamiliar technologies which will be used in an application or during the creation of the application.
- Managerial risk; that which arises from the lack of established management techniques to support new technologies and methods.

These two elements of risk are closely related, and affect one another in a variety of ways. The following sections discuss both elements and suggest some mechanisms for their mitigation.

18.1.1 Technological Risk

Technological risks may take a wide variety of forms, and vary from simple risks to highly complex instances involving the interrelationship of many divergent technologies. Some relatively simple examples of technological risk are:

- Use of a new prototyping tool
- Use of a programming language other than that which is normally used by the particular development organization
- Use of a new programming interface
- Incorporation of new analysis techniques for the gathering of requirements.

The element of risk in the incorporation of new technologies arises from the simple fact that they are new, and likely to be unfamiliar to the majority of development personnel. A period of learning will be necessary, and the probable length of this period must be assessed in the light of required development schedules. Technological risk is always greater where new

techniques and procedures must be evolved in order to support and exercise the new technologies, since elements of managerial risk are then involved. Simple technological tools are relatively easy to learn and use, but techniques for their' effective employment are often learned over a longer period.

The decision as to whether to utilize new technologies in a development project must be based upon the question of whether the development organization possesses the requisite skills to effectively exercise the new technologies, or can acquire such skills within the timeframe of the development project, without adversely affecting the schedule and budget of the project. If the first question can be answered in the positive, there is no risk involved; if not, then it is the second question that constitutes the element of risk.

Technological risk may be mitigated by ensuring that development personnel possess the necessary skills to effectively utilize the new technologies, through the provision of relevant education. The potential benefit to be gained from the use of these technologies must be identified and quantified, in terms of enhanced application functionality, reduced development time and cost, etc. This benefit must be weighed against the time and cost involved in training personnel to a sufficient skill level in order to effectively mitigate the risk, and against the schedule and budget imposed upon the development organization.

Technological risk may also be mitigated by minimizing the managerial risk involved in the use of new technologies. If established managerial techniques can be adapted and applied to the implementation of new technologies, the use of these mechanisms may provide a greater degree of control over the development process, and help to control and reduce the associated technological risk. Thus it can be seen that technological and managerial risk are closely related and complimentary to one another.

18.1.2 Managerial Risk

Managerial risk is somewhat more complex than technological risk, since it involves the effective administration of and control over the use of new technologies. While it is possible to train or employ development personnel in order to gain the required skills in the use of new technologies, it is less easy to obtain the managerial skills necessary to ensure the maximum benefit is gained from their use.

Like technological risk, managerial risk may also vary from the relatively simple to the highly complex. Some examples of managerial risk are:

- Transformation of new analysis techniques into application design specifications
- Establishment of effective control procedures for a new development environment
- Measurement of programmer productivity when using a new programming tool or language.

The managerial risk arises not from the question of whether sufficient skills are present to utilize new technologies, but from the question of whether managerial personnel are sufficiently well-versed in the concepts underlying these technologies to provide effective administration and control over the use of the technologies, in order to ensure that maximum benefit is gained from their use.

Managerial risk may be mitigated by ensuring that managerial personnel possess a sufficient grounding in the principles underlying new technologies, in order that they may successfully adapt existing managerial techniques to the administration and control of the new technologies. These skills may be acquired in a similar manner to the technological skills required by programmers, through training and familiarity with the technologies involved. The decision must be made as to whether the benefit to be gained from the use of these technologies is sufficient to offset the time and effort involved in acquiring the necessary skills and establishing the managerial techniques to effectively control their use.

Managerial risk may also be mitigated by reducing the associated technological risk. For example, a new technology such as object-oriented programming principles can be implemented using tools such as the C programming language and Presentation Manager. There is likely to be a higher degree of familiarity with such tools in the development organization than with tools such as C++ or Smalltalk V. Therefore, implementation of object-oriented principles may be more effectively controlled by the application of established managerial techniques for C application development. While this will not eliminate the element of risk altogether, it will significantly reduce the managerial risk involved, and also help to mitigate the technological risk by facilitating effective control over the development process.

18.2 Configuration/Library Management

While the same techniques of management are valid in both host and workstation environments, the distributed nature of the workstation environment presents difficulties for the centralized control and administration of development resources. This section provides some suggestions as to the ways in which a local area network (LAN) may be used to provide centralized control and administration to a workstation-based application development environment.

In the case of C language applications developed for the Presentation Manager environment, development resources include:

- Application source modules. An application source module is defined to comprise not only the source code itself, but also the local and external interface include files that accompany the source code (see Appendix B, "Application Program Construction").
- Existing code libraries; these may need to be modified or additional routines generated in order to meet the requirements of the application.
- Presentation Manager resources such as icons, fonts, bitmaps and dialog definitions.
- Test data sets or databases.
- Compilers, link-editors, run-time libraries and other development tools; in an evolving development environment, it is crucial to ensure that all development personnel use the same version and modification level of compiler and linkage editor software, and that programming language runtime libraries are consistent in their version.

These resources must be created, tested and placed in locations from which they may be accessed easily and concurrently by a number of application developers, while at the same time maintaining adequate control over their access and particularly over any modifications made to individual modules.

A LAN provides a useful means of enabling access by multiple developers to a common repository of application resources. The ways in which a LAN may be used to address the configuration management issues arising from the PWS development environment are described in the following sections.

18.2.1 Terminology

In the subsequent discussion of configuration and library management, the following terminology will be used. The term **application resource** will be used to indicate a particular development resource such as a source module (along with its supporting include file), a custom-developed dynamic link library (that is, a dynamic link library not taken from a group of generic library modules), a Presentation Manager resource such as an icon or dialog definition etc., which is specific to the current application. Other development resources such as compilers, link-editors, programmers' toolkits, generic code libraries and so on, may be used in the development of an application, but are not considered to be application resources.

The term **production level** will be used to indicate a version of an application resource that has been created, tested and approved for placement in a production library. The process of testing and subsequent approval for placement in a production library is called **baselining**.

The term **user level** will be used to indicate a version of an application resource that is currently undergoing modification, and has not been either tested or placed in a production library. The actual transfer of an application resource from a developer's local work library to a production library is called **promotion**.

18.2.2 Network Organization

The topology of a local area network is very much determined by the structure of the development organization, and by the size and nature of the projects undertaken by that organization. The techniques of LAN installation, configuration and management are beyond the scope of this document. However, it is possible to formulate some simple guidelines which facilitate management of the development process.

Local area networks are typically divided into logical partitions known as **domains**. A domain is defined as a logical subgroup of a network, which contains a defined group of network nodes (machines), a defined set of network resources such as shared disks, directories and printers, and a defined set of authorized users. Each domain thus forms a logical network in its own right, and multiple domains may exist on a single physical network. A domain may include resources residing on multiple network server nodes, and multiple domains may access the resources of any particular server. Each user on the network is provided with a unique user ID and password. An application developer may be defined as an authorized user of multiple domains within the same physical network; the same or different user IDs and passwords may be used.

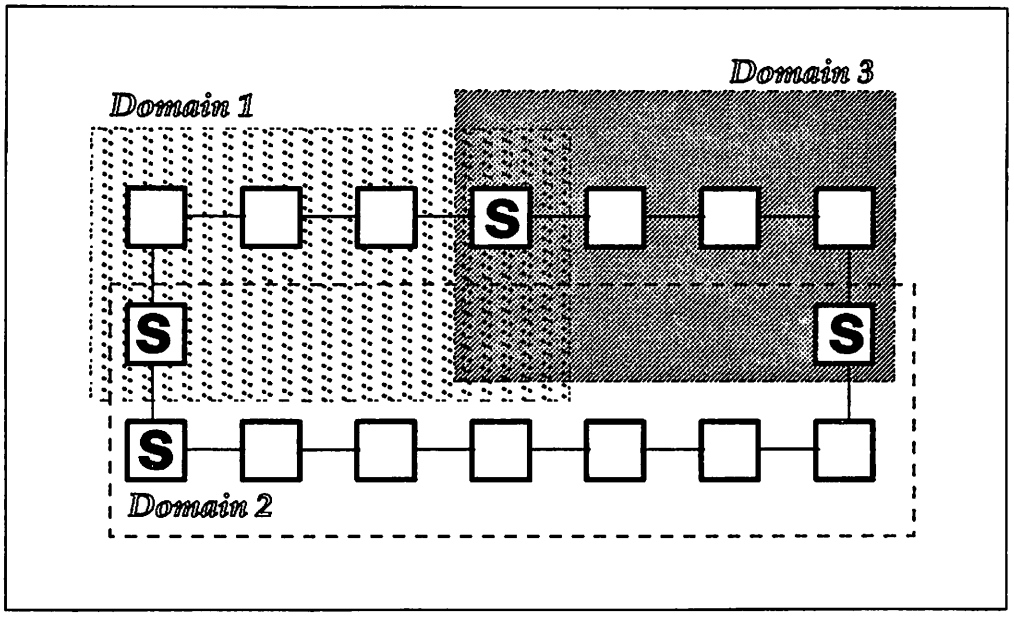

Figure 148. Network Domains

Figure 148 illustrates a network with three domains, each containing a number of network nodes. Each domain has one or more server nodes (marked *S*) upon which reside shared resources accessible by users on other nodes. Note that the servers may be accessed from within a single domain or from multiple domains, and that other network nodes may also belong to multiple domains. Note also that there is no direct mapping between a network node and a network user; a user may, in principle, sign on and access server resources from any network node in the domain.

Assuming a project-team orientation in the development organization, it is expedient to logically group the members of a particular project team, in order that they may be treated as a distinct group and separated from other project teams for purposes of resource access and administration.

In the simplest case, a number of project teams in the same physical location would use the same physical network, partitioned into separate domains for each project team. Each domain would possess its own set of production libraries for application resources; other development resources such as compilers and generic code libraries, which are common across the entire development organization, would be stored in a production library on a single server, and defined within all domains. This technique provides isolation of application resources while also allowing common access to other development resources, and eases the task of maintaining and updating these common resources since only one copy need exist on the network.

The principle of one domain per project team is obviously a "rule of thumb" and must be evaluated in light of the individual development organization. In the case of very large projects, it may be necessary to subdivide the project team into manageable subgroups. This would probably be necessary purely for managerial purposes, irrespective of whether a LAN were to be used. Conversely, very small project teams may not warrant the effort of establishing a separate domain, and several such small teams may be combined in a single management unit with a single network domain.

18.2.3 Common Access to Resources

On a local area network server node, directories may be created that act as production libraries. Production libraries for a particular development project may exist on one or more network server nodes. depending upon the size and organization of the LAN.

These libraries serve as repositories for the current production-level versions of all development resources. The exact number and type of libraries created is highly dependent upon the structure of the development organization and the application under development, but the following skeleton structure is recommended.

```
Server Root Directory ──┬── Compilers and link-editors
                        ├── Application source code
                        ├── Application include files
                        ├── Presentation Manager resources
                        ├── Installation code libraries
                        └── Test data
```

Figure 149. Production Libraries on a LAN Server

All application developers should be given read-only access to production libraries. This will enable those developers to access compilers, link-editors and programming language run-time libraries, and to access the current production-level versions of application source modules, Presentation Manager resources and test data, but *not* to update those production versions.

Application resources currently undergoing modification (that is, user level resources) are held in a work directory on the developer's own workstation, from which only that developer may access them. This restriction of access is implicit since only appropriately declared and configured server nodes may share their disks and directories on the network.

Production level application resources may be transparently accessed at compile or link-edit time by ensuring that each developer's compiler search path specifies the production libraries. The search path should first specify an application developer's local work directory, in order to pick up any user level resources currently being worked upon by that developer, and then search the appropriate production libraries in order to pick up production level copies of other resources not currently subject to modification by that developer. This technique ensures that each application build accesses the latest tested and baselined versions of all application resources, except for those resources that exist in the developer's local work directory, and that are therefore likely to be under test.

Each application resource should have an *owner* appointed at the start of development. This owner may be the application developer primarily responsible for the creation of a source module, or in the case of larger and more complex development projects, may be a developer responsible for the

testing of a number of modules that together comprise a coherent code unit. In either case, the owner is given update access to the files in the production libraries that comprise the module or modules under his/her jurisdiction, and *only* to those files. The updating of each application resource may then be achieved in a controlled manner.

This assumes that the name of the file or files containing each application resource is known at the outset of development. This in turn requires a sound approach to application design and to the correct partitioning of the application at the design stage. It also requires the adoption of a set of file or data set naming conventions across the development organization.

18.2.4 Update/Modification of Resources

When an application developer wishes to make a modification to an application's resources, those resources should be copied from the production library to the developer's local work directory. This is known as **drawdown**, and must be recorded, including the date, time, the identity of the developer and the name of the module being copied. This recording process is known as **checkout**. Only one developer at any time must be allowed to check out and draw down a particular module, in order to avoid the problems inherent in the well-known simultaneous update situation. Control over checkouts may be achieved in a number of ways. However, it is recommended that the drawdown process be achieved by way of a simple utility application that performs the following steps:

1. Accesses a central database that contains the name of each application resource under development, and determines whether that resource is currently checked out.
2. If the resource has been checked out, the utility application returns the identity of the developer by whom the resource has been checked out
3. If the resource has not been checked out, the utility application performs a checkout operation, recording the identity of the application developer and that of the application resource in the central database, and draws down the application resource into the developer's local work directory, ready for modification.

When modification of a resource is complete and the resource has been adequately tested, the user level version should be passed to the resource's owner who, after determining that appropriate testing has been satisfactorily carried out, should then promote the new version to production level, ready for access by other developers.

When a user level version of an application resource is consigned to the resource owner, that version must be deleted from the developer's local work directory, in order to ensure that only the latest production level version is accessed by the compiler or link editor during the next build.. This operation may be automated as part of the promotion process, or may be left as an explicit task for the developer.

18.2.5 Administration

As in any multi-user environment, some degree of system administration is necessary in a LAN environment. When a LAN is used to provide a workstation application development platform, the LAN administrator must perform the following duties:

- At the outset of development for a particular application, create a network domain for that development project. This is optional, and it may be considered expedient to combine a number of smaller projects into a single' network domain.
- Create the production libraries for the project's network domain.
- Register each developer as a user of the project's network domain, and define each developer to have read-only access to production libraries.
- Provide the appointed owner of each application resource with write access to the resource or resources under his/her jurisdiction.
- In the case where an application resource has been checked out and remains checked out for an unnecessary length of time (for example, a developer checks out a resource and then goes on vacation without first submitting the new version for promotion), provide an override capability to cancel the checkout.

It is recommended that the LAN administrator be the same person responsible for administration of the production libraries that contain system software such as compilers and link-editors, in order to place all such system administration responsibilities with the same person.

18.3 Summary

The management and mitigation of risk during the application development process is an important aspect of managing application development, particularly where new or unfamiliar technologies are to be used. There are two closely related elements of risk that arise from the incorporation of new technologies. These are *technological risk*, which arises from the need for adequate skills to exercise the technologies, and *managerial risk*, which arises from the need for adequate administration and control mechanisms to ensure that maximum benefit is gained from the use of the technologies. It is the responsibility of a development manager to assess, quantify and weigh the potential benefits of new technologies against the risk involved in their use, and to provide adequate mitigation of these risks.

The issue of configuration management in a distributed, workstation-based development environment is another issue that must be addressed in order to support the large-scale development of workstation-based applications. The use of a local area network as a development platform for such applications has a number of benefits, particularly from the viewpoint of configuration management and control over application resources such as source code, Presentation Manager resources, test data and the like. The use of a LAN provides:

- Common access by all developers to production level versions of application resources
- The ability to directly access these production level versions during the build process
- The ability to combine production level versions with user level versions of application resources during the build process
- The ability to manage and regulate the modification and update of production level application resources.

The proper use of a LAN in the workstation-based development environment, and the achievement of the aforementioned benefits, requires the adoption of

and adherence to a consistent set of standards in the areas of module naming, access and testing, and a measure of discipline on the part of application developers. However, it is considered that the increased time and effort expended in maintaining these standards is more than offset by the reduced incidence of error and wastage of development time and effort imposed by the lack of adequate coordination and control in a multi-developer project.

The issues of management and control in the workstation-based development environment are of increasing importance as organizations begin to develop and deploy line-of-business applications on workstation platforms. These issues may be adequately addressed by the use of appropriate tools and the adaptation and application of existing management techniques. With the proper care and planning, the maximum benefit may be obtained from the use of the workstation as a development and delivery platform for business applications.

Appendix A. Naming Conventions

It is often a desirable practice to implement common naming conventions for application routines and for symbolic constant and variable names. The adoption of such techniques facilitates application readability and code maintenance, since the nature and role of various routines and data items used by an application may be more easily understood. This chapter proposes some naming conventions; it is not suggested that application developers should use these conventions slavishly, but that they should use the suggestions provided as guidelines in developing their own consistent set of organizational standards.

The conventions proposed herein will cover the following areas:

- Symbolic names and constants
- Subroutine names
- Window and dialog procedure names
- Variable names.

The conventions proposed will adhere to most "standard" C programming conventions, in that lowercase characters will be used for routine and variable names, with uppercase characters used for symbolic names and constants. Application developers wishing to use standardized naming conventions for applications written in other languages will obviously need to adapt these conventions to suit their particular language implementation.

The conventions proposed herein will also use a notational concept known as **Hungarian notation**, named for its inventor, Charles Simyoni. Under this notational system each variable, symbolic name or procedure name is prefixed by a one-, two- or three-character mnemonic that indicates its type or function.

A.1 Symbolic Names and Constants

In a PM application, symbolic names are typically used within an application to represent numeric values such as message classes or control window identifiers by a meaningful name rather than a less meaningful numeric representation. A list of suggested prefixes is given in Table 6, to give an indication of type when using a symbolic name or constant.

Table 6. Type Prefixes for Symbolic Constants

Item	Constant Type	Prefix
Menu Item (Command)	Integer	MI_
Check Box	Integer	CK_
Entry Field	Integer	EF_
List Box	Integer	LB_
Push Button	Integer	PB_
Radio Button	Integer	RB_
Static Text String	String	STR_
Window Class	String	WC_
Dialog Class	String	DC_
Message Class (Application-defined)	Integer	WMP_

These one-, two- or three-letter prefixes may be concatenated with the actual symbolic names of control windows, window classes etc., in order to provide a more meaningful representation of the symbolic name in the source code.

A.2 Subroutine Names

For the purpose of discussion, a distinction will be made between subroutines invoked using normal programming language calling mechanisms, and window or dialog procedures invoked by Presentation Manager in response to the application issuing a **WinDispatchMsg()**, **WinPostMsg()** or **WinSendMsg()** call. Window and dialog procedures are discussed in the following section.

When examining application code, it is useful to know the type of event handled or processing carried out by a particular function or subroutine, without the need for detailed examination of the code for that subroutine. This capability can be facilitated by the use of a prefix to the function or subroutine name, which indicates the type of the function or subroutine to the reader.

Since the types of functions carried out within applications may be extremely diverse, no standards will be suggested here. However, readers should note the potential benefits of such a practice, and may wish to adopt a suitable convention for their own applications.

A.3 Window and Dialog Procedure Names

In order to indicate that a particular subroutine within an application is a window procedure, it is suggested that all window procedure names should be prefixed with *wp* in lowercase letters. Similarly, dialog procedures for processing modal dialog boxes, should be prefixed with "dp" to identify the nature of their processing and to differentiate them from normal window procedures.

A.4 Variable Names

It is far easier to determine the nature and usage of a variable if its data type is known to the reader. Variable names may be prefixed with mnemonics indicating their data type, in a similar way to that proposed for symbolic names and constants. A list of suggested prefixes for various data types is given in Table 7.

Table 7 (Page 1 of 2). Type Prefixes for Variables

Data Type	Definition	Prefix
Boolean	BOOL (flag)	f
Character	CHAR	ch
Unsigned character	UCHAR	uch
String	CHAR[]	sz
Short integer	SHORT	s
Unsigned short integer	USHORT	us
Long integer	LONG	l
Unsigned long integer	ULONG	ul

Table 7 (Page 2 of 2). Type Prefixes for Variables		
Data Type	**Definition**	**Prefix**
Handle	HWND, HMODULE, etc	h

For example, a character string variable (a zero- or null-terminated string) named *WindowTitle* might have an attached mnemonic prefix of *sz* to indicate the data type, making the variable name *szWindowTitle*. This is a simple example; to take a more complex instance, a handle to a window might have a variable name *hMainWindow*, which would differentiate it in the source code from a window procedure *wpMainWindow* or other data items relating to the window, while maintaining an indication of the relationship between the items by the similarity in their names.

Prefixing variable names in this way has the additional advantage that a compiler cross-reference listing will group together all variables of the same data type. Any redundancies may thus be seen at a glance.

A pointer to a variable is indicated by using an additional prefix *p* before the prefix indicating the data type of the variable. Some examples are shown in Table 8.

Table 8. Type Prefixes for Pointers		
Data Type	**Definition**	**Prefix**
Pointer to CHAR	CHAR *	pch
Pointer to string	PSZ	psz
Pointer to function	PFN, PFNWP	pfn

As a further example, an unsigned integer *UserResponse* might have a prefix of *us* making the variable name *usUserResponse*. A pointer to this variable would have the name *pusUserResponse*. The name thus indicates both the data type of the pointer and its relationship with the variable to which it points.

Appendix B. Application Program Construction

This section of the document presents guidelines for the structuring of applications and their component modules, in order to achieve the optimum level of modularity and granularity within an application. The guidelines contained herein are particularly applicable to the C programming language, although they may also apply to other languages with similar structures.

B.1 Modularization

Within a Presentation Manager application, it is recommended that the application code be separated into distinct source modules, as follows:

- Each window procedure (that is, application object) should be placed in its own separate source module, along with functions and subroutines created for and exclusively called by that window procedure. This creates the situation where a single window is contained per source module, which preserves isolation and facilitates independence of application objects.
- Type definitions, variable and constant declarations (including private message classes) and function prototypes that are local to a particular source module should be placed in a *private header file* or alternatively, included in the source module itself.
- Function prototypes for those window procedures or subroutines that will become the entry points to the source module should be placed in a separate header file, along with type definitions, variable and constant declarations that will be required by other source modules calling those procedures or subroutines. This header file may then be referenced by each source module that requires access to these procedures or routines. This header file is known as the *public header file*.
- Global type definitions, variable and constant declarations should be placed in a *global header file* that may be referenced by each source module. In the ideal case, global variables and constants should not be used by an application, and this header file would therefore not be required.
- Generic functions and subroutines accessed by more than one window procedure should be placed in a separate source module with its own header file (known as a *generic routines header file*), which may be referenced by each module requiring access to the generic routines. This includes "functions" such as dialog definitions and dialog procedures that are accessed from multiple window procedures, and message handling routines.
- Presentation Manager resources used by modules within an application should be placed in a resource script file, and their associated identifiers defined in the application's public header file.

Note that a Workplace Shell object's source code is automatically partitioned in a similar manner to that described above, since the various files are created by the SOM Precompiler from the object's class definition file.

Separation of the application code into its constituent application objects in this manner facilitates the isolation and independence of application objects, and enhances the potential for their subsequent reuse. It also eases the task of application maintenance and change management, by effectively modularizing

an application and therefore helping to contain the scope of change within a single source module and its dependent header files.

The separation of header files into private, public and global definitions in the manner described above further enhances the independence of application objects and facilitates change management, in the following ways:

- The separation of an application object's public interfaces means that other application objects are aware only of those interfaces, and not of the internal definitions and operations of the application object. Changes within an application object, to local type, variable or constant definitions, do not impact other application objects with which the changed object communicates
- The separation of the private and public interfaces explicitly defines each of these interfaces, so that a maintenance programmer is clearly aware of changes that will or will not impact other application objects. Other applications that do require modification as a result may be easily identified, since their source modules will contain a reference to the changed object's external interface header file.

When managing the development of large projects, an "owner" should be appointed for each source module. This owner is typically a member of the development team who bears responsibility for that module. A module's owner should also have the responsibility for that module's private and public header files.

B.2 Header Files

Header files should be used wherever possible in order to isolate data and function declarations from the application code, thereby enhancing modularity and improving readability.

For management purposes and to facilitate subsequent maintenance of the application code, a header file should include a prolog identifying the application and source module with which it is associated, its author and owner, and whether the header file is a private, public, global or generic header file (see below for definitions of these terms).

B.2.1 Private Header File

Each source module should have its own private header file, or have the contents of such a file included in the source module itself. The private header file should itself include:

- Declarations for all local constants used within the source module; that is, those constants that are not accessed or referenced from outside the source module. This includes those application-defined message classes used by a window procedure in indicating events to itself, or by dependent functions and/or subroutines to indicate messages to their parent window procedure.
- Declarations for all non-local variables used within the source module; that is, those variables that are accessed from more than one routine within the source module, but are not accessed from outside the source module.
- Prototypes for all functions and subroutines that are accessed only from within the source module.

A private header file should be referenced, using an appropriate *#include* statement, only by its own source module.

B.2.2 External Interface Header File

Each source module should possess its own public header file. The public header file should contain:

- Function prototypes for the entry-point functions or subroutines within the source module, and *only* for those routines. This preserves the isolation of the source module's internal workings from the calling application.
- Type definitions for any application-defined data types required by the entry-point functions or subroutines.
- Message class definitions for application-defined message classes that will be used to signal events to a window procedure within the source module.

A public header file should be referenced by its own source module and by any other source module which requires access to the entry points of the module. In an ideal case, where optimum isolation is achieved by relating all processing within the source module to a single window, access to these entry points would be achieved by:

- A single "conventional" subroutine call to create the window, with the caller specifying appropriate parameters and the called routine returning a handle to the window. The public header file must therefore contain definitions for any application-defined data types to be passed as parameters to this call.
- Passing a series of messages to the window in order to indicate events and request actions. The public header file must therefore also contain definitions for any application-defined message classes used by the window.

Since the public header file contains the interfaces to its parent window (that is, application object) it should be carefully documented, and the entry points and their means of access should be placed in the application's design documentation. Interfaces to those application objects that are identified as having potential for reuse should also be placed in an **Interface Control Document** from which they may be accessed for future reuse of the application object (see B.4, "Packaging" on page 333).

B.2.3 Global Header File

An application may contain a global header file, which itself should contain:

- Declarations for all global constants used by the application; that is, those constants that are accessed from more than one source module within the application.
- Declarations for all global variables used by the application; that is, those variables that are accessed from more than one source module within the application. Note that this does not include those variables that are used by multiple routines within the same source module.

The global header file should be referenced, using an appropriate *#include* statement, by all source modules within the application, other than the module(s) containing generic routines.

B.2.4 Generic Routines Header File

One or more source modules in an application may contain generic routines that are accessed from multiple source modules within the application. These source modules may possess their own local header files to define data and functions accessed only from within their own module. In addition, such source modules should possess a single generic routines header file per application, which should contain:

- Prototypes for all generic functions and subroutines that will be accessed from other source modules within the application
- Declarations for data types and constants necessary to the invocation of these generic routines.

The generic routines header file should be referenced, via an appropriate *#include* statement, by its own source modules and by each source module within the application that requires access to generic routines.

B.2.5 System-Supplied Header Files

Source modules in an application will typically require access to operating system or C language functions. Prototypes for these functions and declarations for their associated data types and constants are provided in system-supplied header files. Examples of system-supplied header files are the OS/2 system functions file *os2.h* and the C language run-time library files such as *stlib.h* and *string.h*.

B.3 Data Abstraction and Encapsulation

In order to enhance reusability and to facilitate the containment (and therefore the management) of change, data definition and initialization should, wherever possible, be encapsulated within source modules as follows:

- Local variables should be used wherever possible. Where the value of a variable must be held between invocations of the same function or subroutine, the *static* keyword may be used in declaring the variable.
- For window procedures where the values of variables must be held beyond the scope of processing a single message, a memory object may be allocated for the variable(s) and a pointer to that object stored in the window words of the window.
- External data objects such as files, databases, etc., should be defined and accessed only from within a single window procedure and its dependent functions and subroutines. Definition of and/or establishment of access to such data objects should be performed upon window creation as part of the WM_CREATE message processing, and termination of access should be performed upon window closure as part of the WM_CLOSE message processing.
- The use of global variables should be avoided wherever possible, although it is recognized that global variables are necessary in certain circumstances within a Presentation Manager application. Where global variables are to be used, they should be declared in the application's main source module (that is, the module that contains the application's main routine) and referenced from the application's global header file using the *extern* keyword.
- Except in the case noted above, the use of external variable declarations using the *extern* keyword should be avoided wherever possible, since this

creates an interdependence between source modules (and therefore between application objects) that may subsequently limit the potential for reuse of those objects. Variables should preferably be defined locally within a source module or alternatively, defined globally and referenced in the application's global header file.

- Where non-local variables and/or constants (that is, variables and/or constants that are accessed only from within a particular source module, but are not local to any routine within that module) are declared, they should be placed in that source module's private header file or, if the module does not possess its own header file, placed at the beginning of the source module.

The practice of maximizing the use of local and encapsulated data, and of minimizing and simplifying the external interfaces of application objects, will achieve the maximum level of isolation and therefore of independence between application objects, thus enhancing the potential for their reuse and facilitating their future maintenance by isolating their internal workings from those of other application objects.

B.4 Packaging

As already mentioned in B.1, "Modularization" on page 329, application code should be separated into distinct source modules. These source modules are then compiled to produce individual object modules, which in turn are link-edited to produce executable code. However, the executable application code may itself consist of more than one executable module, by virtue of the dynamic linking capabilities of the OS/2 operating system.

B.4.1 Application Object Modules

If the foregoing guidelines are followed, each object module produced by the C language compiler will contain the following:

- Definitions for all locally defined data types, variables and constants required by routines within the module, obtained from the module's private header file at compile time
- Prototypes for all local window procedures, functions and subroutines accessed only by routines within the module, obtained from the application's private header file at compile time
- Definitions for all external data structures, variables and constants required to communicate with other source modules, obtained from one or more public header files at compile time
- Prototypes for all external window procedures, functions and subroutines accessed by routines within the module, other than those that constitute generic routines, obtained from one or more public header files at compile time
- Declarations for all global variables and constants required by routines within the module, obtained from the application's global header file at compile time
- Prototypes for all generic routines, obtained from the application's generic routines header file at compile time
- Code for those routines contained within the module.

Each object module therefore exists as a coherent identity in its own right, and ideally has no dependence upon other object modules, other than the need to communicate with and make use of window procedures, functions and subroutines contained within those modules, which is achieved through public interfaces that are clearly defined and documented. The separation of source modules is now carried over to the object code, in that each object module represents a separate application object.

It is therefore possible to assemble a number of object modules in various ways to achieve an executable application. The following guidelines are offered for the construction of the executable application.

B.4.2 Application Executable File

The application's main executable file (that is, the file that is invoked to start the application) should contain those object modules which are required to execute the application, and that do not contain generic routines or generic Presentation Manager resources.

Where it is envisaged that an application object is usable only by the current application, and has no possible potential for future reuse, it may be bound with the application's main executable file. However, if such potential for reuse exists, the application object should be placed in a dynamic link library.

B.4.3 Dynamic Link Libraries

The following routines and objects should, wherever possible, be placed in dynamic link libraries:

- Those application objects that have been identified as having potential for future reuse by other applications
- Those routines identified as generic routines, which will be accessed from more than one window procedure
- Those Presentation Manager resources (for example, dialog definitions, along with their associated dialog procedures) that are accessed from multiple window procedures within the application.

The placement of such items in dynamic link libraries enhances their independence from the application's code, enabling changes to be made without affecting the code or requiring application maintenance. In addition, placement in dynamic link libraries facilitates reuse of such items by multiple applications. It is thus possible to achieve object code reuse at both the application object and at the subroutine level.

To this end, all items placed in dynamic link libraries should have their external interfaces documented, baselined and included in an Interface Control Document, for subsequent use as development library routines. It is the responsibility of the development organization to establish procedures for the testing and acceptance of reusable code modules into libraries from which they may subsequently be accessed by application developers, and the creation of a repository of documentation detailing the application objects that are available, their behavior and external interfaces.

Appendix C. OS/2 Kernel API Functions

This section of the document describes the functions that are available to application programs for accessing operating system services. Note that only the operating system kernel functions are described in this section, since the Presentation Manager programming interface has not significantly changed, although a number of new Presentation Manager functions have been added.

The following tables list the kernel functions available under OS/2 Version 1.3 and their equivalent function under OS/2 Version 2.0, categorized by function area.

C.1 Memory Allocation and Management

Memory management under OS/2 Version 2.0 is greatly simplified through use of the 32-bit flat memory model, eliminating the need for applications to create and manipulate separate memory segments.

Table 9 (Page 1 of 2). Memory Management Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 memory allocation and management functions.

16-Bit Function Name	*32-Bit Function Name*
DosAllocSeg	N/A
DosAllocShrSeg	N/A
DosGetShrSeg	N/A
DosGetSeg	N/A
DosGiveSeg	N/A
DosReallocSeg	N/A
DosFreeSeg	N/A
DosAllocHuge	N/A
DosGetHugeShift	N/A
DosReallocHuge	N/A
DosCreateCSAlias	N/A
DosLockSeg	N/A
DosUnLockSeg	N/A
DosMemAvail	N/A
DosSizeSeg	N/A
DosGetResource	DosGetResource
DosSubAlloc	DosSubAlloc
DosSubFree	DosSubFree
DosSubSet	DosSubSet
N/A	DosSubUnSet
N/A	DosAllocMem
N/A	DosAllocSharedMem
N/A	DosGetNamedSharedMem
N/A	DosGetSharedMem

Table 9 (Page 2 of 2). Memory Management Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 memory allocation and management functions.	
16-Bit Function Name	***32-Bit Function Name***
N/A	DosGiveSharedMem
N/A	DosFreeMem
N/A	DosSetMem
N/A	DosQueryMem

C.2 Session Management

There are few changes in session management functions between OS/2 Version 1.3 and OS/2 Version 2.0. Only the **DosSMRegisterDD()** function has been removed from OS/2 Version 2.0.

Table 10. Session Management Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 session management functions.	
16-Bit Functions Name	***32-Bit Function Name***
DosStartSession	DosStartSession
DosSetSession	DosSetSession
DosSelectSession	DosSelectSession
DosStopSession	DosStopSession
DosSMRegisterDD	N/A

C.3 Task Management

A number of function names have changed in the task management area. These changes are primarily due to the simplified 32-bit programming environment.

Table 11 (Page 1 of 2). Task Management Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 task management functions.	
16-Bit Function Name	***32-Bit Function Name***
DosCreateThread	DosCreateThread
DosCWait	DosWaitChild
N/A	DosWaitThread
DosResumeThread	DosResumeThread
DosSuspendThread	DosSuspendThread
DosEnterCritSec	DosEnterCritSec
DosExecPgm	DosExecPgm
DosExit	DosExit
DosExitCritSec	DosExitCritSec
DosExitList	DosExitList
DosGetInfoSeg	N/A
N/A	DosGetInfoBlocks
DosGetPrty	N/A
DosKillProcess	DosKillProcess

Table 11 (Page 2 of 2). Task Management Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 task management functions.	
16-Bit Function Name	***32-Bit Function Name***
N/A	DosKillThread
DosSetPrty	DosSetPriority
DosGetPID	N/A
DosGetPPID	N/A
DosR2StackRealloc	N/A
DosCallBack	N/A
DosRetForward	N/A
N/A	DosDebug
DosPTrace	N/A

The following comments apply to the functions described in the table above:

- The **DosCreateThread()** function is enhanced in Version 2.0 to allow dynamic stack growth, operating system management of stacks for secondary threads, and the ability to create a suspended thread.
- The *InfoSeg* architecture of 16-bit OS/2 versions is redefined for the flat memory model. The equivalent per thread and per process data is obtained under Version 2.0 using the **DosGetInfoBlocks()** function, whereas the non-changing values can be obtained using the **DosQuerySysInfo()** function.
- The **DosGetPrty()**, **DosGetPID()**, and **DosGetPPID()** functions are not implemented since this information is available through the **DosGetInfoBlocks()** function.
- **DosPTrace()** support is provided under OS/2 Version 2.0 for 16-bit debugging tools such as CodeView. For 32-bit debugging tools, the **DosPTrace()** function is replaced by the **DosDebug()** function.
- **DosWaitThread()** is a new function allowing a thread to suspend itself, waiting for other threads within a process to terminate.
- **DosKillThread()** is a new function allowing a thread to forcibly terminate another thread within the current process.
- The **DosR2StackRealloc()**, **DosCallBack()**, and **DosRetForward()** functions are not applicable in the 32-bit flat memory model environment.

C.4 Signal and Exception Handling

OS/2 Version 2.0 removes the function handling signals and combines signal handling with hardware exception handling, providing a more unified approach that allows greater flexibility.

Table 12 (Page 1 of 2). Exception Handling Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 signal and exception handling functions.	
16-Bit Function Name	***32-Bit Function Name***
DosHoldSignal	N/A
DosSetSignalHandler	N/A
DosSendSignal	N/A

Table 12 (Page 2 of 2). Exception Handling Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 signal and exception handling functions.	
16-Bit Function Name	***32-Bit Function Name***
N/A	DosSetKBDSigFocus
DosSetVec	N/A
N/A	DosSetExceptionHandler
N/A	DosUnSetExceptionHandler
N/A	DosRaiseException
N/A	DosUnwindException

The **DosSetKBDSigFocus()** function is used to allow 32-bit programs to inform the operating system that they are "primed for signals", although the signals are dispatched as exceptions.

Note the following:

- 16-bit **DosSetVec()** exceptions are supported under OS/2 Version 2.0.
- The 32-bit OS/2 Version 2.0 exception manager allows per-thread exception handling.
- OS/2 Version 2.0 does not allow an exception handler to be registeed for the numeric processor exception (NPX).

C.5 Interprocess Communication

OS/2 Version 2.0 provides the same interprocess communication and synchronization facilities as OS/2 Version 1.3. Differences in functions used to invoke these facilities are shown in the following tables.

C.5.1 Anonymous Pipes

The name of the **DosMakePipe()** function has been changed to **DosCreatePipe()** to conform with the consistent naming rules introduced in OS/2 Version 2.0.

Table 13. Anonymous Pipe Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 anonymous pipe functions.	
16-Bit Function Name	***32-Bit Function Name***
DosMakePipe	DosCreatePipe

Other functions used to access anonymous pipes are basic file I/O functions, and are described in C.10, "File I/O" on page 342.

C.5.2 Named Pipes

Changes to the functions used to create and manipulate named pipes have been made in order to conform to the consistent naming rules introduced in OS/2 Version 2.0.

Table 14. Named Pipe Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 named pipe functions.	
16-Bit Function Name	***32-Bit Function Name***
DosCallNmPipe	DosCallNPipe
DosConnectNmPipe	DosConnectNPipe
DosDisConnectNmPipe	DosDisConnectNPipe
DosMakeNmPipe	DosCreateNpipe
DosPeekNmPipe	DosPeekNPipe
DosQNmPHandState	DosQueryNPHState
DosQNmPipeInfo	DosQueryNPipeInfo
DosQNmPipeSemState	DosQueryNPipeSemState
DosRawReadNmPipe	DosRawReadNPipe
DosRawWriteNmPipe	DosRawWriteNPipe
DosSetNmPHandInfo	DosSetNPHState
DosSetNmPipeSem	DosSetNPipeSem
DosTransactNmPipe	DosTransactNPipe
DosWaitNmPipe	DosWaitNPipe

C.5.3 Queues

No changes have been made to function names for queue manipulation between the OS/2 Version 1.3 and OS/2 Version 2.0 implementations.

Table 15. Queue Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 queue management functions.	
16-Bit Function Name	***32-Bit Function Name***
DosCreateQueue	DosCreateQueue
DosOpenQueue	DosOpenQueue
DosCloseQueue	DosCloseQueue
DosPeekQueue	DosPeekQueue
DosPurgeQueue	DosPurgeQueue
DosQueryQueue	DosQueryQueue
DosReadQueue	DosReadQueue
DosWriteQueue	DosWriteQueue

The queueing functions for the 16-bit and 32-bit environments are virtually identical; the only difference is that the 32-bit API uses 0:32 addressing and may use element sizes greater than 64KB. Note that a a 32-bit application *may not* open a queue created by a 16-bit application using the 16-bit **DosCreateQueue()** function.

C.5.4 Semaphores

Significant changes have been made to semaphore functions under OS/2 Version 2.0, in order to provide additional flexibility and enhance the architectural independence of applications using semaphores.

Table 16. Semaphore Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 semaphore functions.

16-Bit Function Name	***32-Bit Function Name***
DosSemClear	N/A
DosSemRequest	N/A
DosSemSet	N/A
DosSemSetWait	N/A
DosSemWait	N/A
DosMuxSemWait	N/A
DosCloseSem	N/A
DosCreateSem	N/A
DosOpenSem	N/A
DosFSRamSemRequest	N/A
DosFSRamSemClear	N/A
N/A	DosCreateMutexSem
N/A	DosOpenMutexSem
N/A	DosCloseMutexSem
N/A	DosRequestMutexSem
N/A	DosReleaseMutexSem
N/A	DosQueryMutexSem
N/A	DosCreateEventSem
N/A	DosOpenEventSem
N/A	DosCloseEventSem
N/A	DosResetEventSem
N/A	DosPostEventSem
N/A	DosWaitEventSem
N/A	DosQueryEventSem
N/A	DosCreateMuxWaitSem
N/A	DosOpenMuxWaitSem
N/A	DosCloseMuxWaitSem
N/A	DosWaitMuxWaitSem
N/A	DosAddMuxWaitSem
N/A	DosDeleteMuxWaitSem
N/A	DosQueryMuxWaitSem

OS/2 Version 2.0 provides 16-bit entry points allowing compatibility with 16-bit applications that use the old system semaphores, RAM semaphores and Fast Safe RAM (FSR) semaphores.

C.6 Message Retrieval

Few changes have been made to the kernel message retrieval functions. Only the name of the **DosInsMessage()** function has been changed to **DosInsertMessage()** in order to conform with the consistent naming rules introduced in OS/2 Version 2.0.

Table 17. Message Retrieval Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 message retrieval functions.

16-Bit Function Name	***32-Bit Function Name***
DosGetMessage	DosGetMessage
DosInsMessage	DosInsertMessage
DosPutMessage	DosPutMessage

C.7 Timer Services

A number of timer function names have changed under OS/2 Version 2.0, in order to conform to the consistent naming rules introduced in OS/2 Version 2.0.

Table 18. Timer Services Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 timer services functions.

16-Bit Function Name	***32-Bit Function Name***
DosGetDateTime	DosGetDateTime
DosSetDateTime	DosSetDateTime
DosSleep	DosSleep
DosTimerAsync	DosAsyncTimer
DosTimerStart	DosStartTimer
DosTimerStop	DosStopTimer

C.8 Dynamic Linking

A number of dynamic linking function names have been changed in order to conform to the consistent naming rules introduced in OS/2 Version 2.0.

Table 19 (Page 1 of 2). Dynamic Linking Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 dynamic linking functions.

16-Bit Function Name	***32-Bit Function Name***
DosLoadModule	DosLoadModule
DosFreeModule	DosFreeModule
DosGetProcAddr	DosQueryProcAddr
DosGetModHandle	DosQueryModuleHandle
DosGetModName	DosQueryModuleName
DosQAppType	DosQueryAppType
DosGetMachineMode	N/A
BadDynLink	N/A
DosGetVersion	N/A

Table 19 (Page 2 of 2). Dynamic Linking Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 dynamic linking functions.	
16-Bit Function Name	***32-Bit Function Name***
DosGetEnv	N/A

C.9 Device I/O

No changes have been made to device I/O functions under OS/2 Version 2.0.

Table 20. Device I/O Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 device I/O functions.	
16-Bit Function Name	***32-Bit Function Name***
DosBeep	DosBeep
DosCLIAccess	N/A
DosPortAccess	N/A
DosDevConfig	DosDevConfig
DosPhysicalDisk	DosPhysicalDisk

C.10 File I/O

A number of changes have been made to file I/O functions, in order to conform to the consistent naming rules introduced in OS/2 Version 2.0.

Table 21 (Page 1 of 2). File I/O Functions. This table compares OS/2 Version 1.3 and OS/2 Version 2.0 file I/O functions.	
16-Bit Function Name	***32-Bit Function Name***
DosBufReset	DosResetBuffer
DosChDir	DosSetCurrentDir
DosChgFilePtr	DosSetFilePtr
DosClose	DosClose
DosDelete	DosDelete
DosDevIOCTL	DosDevIOCTL
DosDupHandle	DosDupHandle
DosEditName	DosEditName
DosFileIO	DosFileIO
DosFileLocks	DosSetFileLocks
DosFindClose	DosFindClose
DosFindFirst	DosFindFirst
DosFindNext	DosFindNext
DosFindNotifyClose	DosFindNotifyClose
DosFindNotifyFirst	DosFindNotifyFirst
DosFindNotifyNext	DosFindNotifyNext
DosFSAttach	DosFSAttach
DosFSCtl	DosFSCtl

Table 21 (Page 2 of 2). File I/O Functions. This table compares OS/2 Version 1.3 and OS/2 Version 2.0 file I/O functions.	
16-Bit Function Name	***32-Bit Function Name***
DosMkDir	DosCreateDir
DosMove	DosMove
DosNewSize	DosSetFileSize
DosOpen	DosOpen
DosQCurDir	DosQueryCurrentDir
DosQCurDisk	DosQueryCurrentDisk
DosQFHandState	DosQueryFHState
DosQFileInfo	DosQueryFileInfo
DosQFileMode	DosQueryFileMode
DosQFSAttach	DosQueryFSAttach
DosQFSinfo	DosQueryFSInfo
DosQHandType	DosQueryHType
DosQPathInfo	DosQueryPathInfo
DosQSysInfo	DosQuerySysInfo
DosQVerify	DosQueryVerify
DosRead	DosRead
DosReadAsync	N/A
DosRmDir	DosDeleteDir
DosScanEnv	DosScanEnv
DosSearchPath	DosSearchPath
DosSelectDisk	DosSetDefaultDisk
DosSetFHandState	DosSetFHState
DosSetFileInfo	DosSetFileInfo
DosSetFileMode	DosSetFileMode
DosSetFsInfo	DosSetFsInfo
DosSetMaxFH	DosSetMaxFH
DosSetPathInfo	DosSetPathInfo
DosSetVerify	DosSetVerify
DosWrite	DosWrite
DosWriteAsync	N/A

C.11 Code Page Support

Code page support has been simplified under OS/2 Version 2.0, and a number of function names have been changed to conform to the consistent naming rules introduced in Version 2.0.

Table 22. Code Page Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 file code page support functions.	
16-Bit Function Name	***32-Bit Function Name***
DosSetCp	N/A
DosSetProcCp	DosSetProcessCp
DosGetCp	DosQueryCp
DosGetCtryInfo	DosQueryCtryInfo
DosCaseMap	N/A
DosGetDBCSEv	DosQueryDBCSEnv
DosGetCollate	DosQueryCollate

The **DosSetCp()** function is no longer implemented, since under OS/2 Version 2.0 an application may only set the code page for its own process, using the **DosSetProcessCp()** function.

The **DosCaseMap()** function is no longer implemented since the same task is performed by the **WinUpper()** function.

C.12 Error Management

No changes have been made to error management functions under OS/2 Version 2.0.

Table 23. Error Management Functions. This table compares the OS/2 Version 1.3 and OS/2 Version 2.0 error management functions.	
16-Bit Function Name	***32-Bit Function Name***
DosErrClass	DosErrClass
DosError	DosError

Appendix D. Problem Reporting Worksheet

The following page contains a worksheet that may be used for reporting problems encountered during application testing. This worksheet follows the guidelines given in Chapter 16, "Problem Determination," and requires the application user to complete the following information:

- The name of the application. This is necessary in order that developers may search for the error in the correct program.
- The title of the active window when the error occurred. This is necessary in order to identify the window procedure in which the problem most likely resides.
- The title of the last user action prior to the error occurring. This is required in order to identify the Presentation Manager message that is likely to have caused the error; it is probable that the problem lies in the processing for this message.
- Whether the action was attempted for the first time, or whether the action had already been successfully completed one or more times during the current execution of the application. This indicates whether the problem is probably a resource allocation error, or whether another cause is more likely.
- A description of the error, including an error message if one was provided by the application, Presentation Manager or the operating system.
- The name of the user who discovered the error, along with a telephone number for future contact if any clarification is required.
- The date and time at which the error occurred. While not necessary from a problem determination viewpoint, this information allows a problem management coordinator to monitor the amount of time necessary to respond to and rectify application problems, and therefore allows management to determine the effectiveness of the testing and debugging process.

The worksheet also contains a space at the foot of the page, into which readers may insert the name of the person responsible for coordinating problem determination and resolution activities, and to whom completed worksheets should be sent. This person would then typically allocate the isolation, diagnosis and resolution of problems to specific developers.

For a more complete description of Presentation Manager problem determination procedures, readers should refer to Chapter 16, "Problem Determination."

Readers may photocopy the worksheet on the following page, insert the name of the appropriate coordinator, and distribute the worksheet within their organizations for problem-reporting purposes.

Problem Reporting Worksheet		
This worksheet is to be completed by application testers, for all problems encountered during testing of Presentation Manager applications.		
Application: *What was the name of the application you were running when the error occurred?*		
Window: *What was the title of the window you were accessing when the error occurred?*		
Action: *What was the last action bar entry or accelerator key combination you selected?*		
☐ First time action was attempted	☐ Action was repeated multiple times	
Error Description: *Provide a description of the error, including the error message if one was displayed.*		
Reported by	Name:	Ext:
Include the date/time at which the error occurred	Date:	Time:
When completed, please forward this worksheet to:		

Appendix E. Source Code for the PWFolder and PWFinanceFile objects

This appendix details the source code for the PWFolder and PWFinanceFile objects.

E.1 Source Code for the PWFolder Object

This section list the source code necessary to generate the dynamic link library (DLL) file PWFOLDER.DLL.

E.1.1 Source Code for the PWFolder.CSC file

```
##########################################################
#                                                        #
#  PWFOLDER.CSC          (c) IBM Corporation 1992        #
#                                                        #
#  This class derives from WPFolder, and is used to      #
#  represent a folder which is protected by a password.  #
#                                                        #
##########################################################

#
# Include the class definition file for the parent class
#
include <wpfolder.sc>

#
#   Define the new class
#
class: PWFolder,
       file stem = pwfolder,
       external prefix = pwfolder_,
       class prefix = pwfoldercls_,
       major version = 1,
       minor version = 1,
       local;

-- PWFolder is a Password protected folder.
-- It is derived as follows:
--        SOMObject
--          - WPObject
--            - WPFileSystem
--              - WPFolder
--                - PWFolder

#
# Specify the parent class
#
parent: WPFolder;

#
# Specify the release order of new methods
#
release order: LockFolder;

#
# Passthru a debug message box to the .ih file
# (for inclusion in the .c file)
#
passthru: C.h, after;

#define DebugBox(title, text)  WinMessageBox(HWND_DESKTOP,HWND_DESKTOP, \
                               (PSZ) text , (PSZ) title, 0, \
                               MB_OK | MB_INFORMATION )

endpassthru;
```

```
#
# Passthru private definitions to the .ph file
# (for inclusion in the .c file)
#
passthru: C.ph;

typedef struct _PWF_INFO {                      /* Define password structure */
  CHAR   szPassword[20];                        /* Folder current password   */
  CHAR   szCurrentPassword[20];                 /* User-entered password     */
  CHAR   szUserid[20];                          /* Userid                    */
} PWF_INFO;
typedef PWF_INFO *PPWF_INFO;                    /* Define pointer type       */

endpassthru;

#
# Define instance data for the class
#
data:
CHAR szPassword[20];
--  This is the password which locks the folder

CHAR szCurrentPassword[20];
--  This is the password the user has typed in to be
--  checked against the lock password

CHAR szUserid[20];
--  Userid

#
# Define new methods
#
methods:

BOOL QueryInfo(PPWF_INFO pPWFolderInfo), private;
--
--  METHOD:   QueryInfo                                   PRIVATE
--
--  PURPOSE:  Copies instance data into the PWF_INFO structure.
--
--  INVOKED:  From PasswordDlgProc
--

BOOL SetInfo(PPWF_INFO pPWFolderInfo), private;
--
--  METHOD:   SetInfo                                     PRIVATE
--
--  PURPOSE:  Sets instance data from the PWF_INFO structure.
--
--  INVOKED:  From PasswordDlgProc
--

BOOL LockFolder();
--
--  METHOD:   LockFolder                                  PUBLIC
--
--  PURPOSE:  Invalidates the current password, thereby locking the folder.
--
--  INVOKED:  From _wpMenuItemSelected
--

#
# Specify methods being overridden
#
override wpInitData;
--
--  METHOD:   wpInitData                                  PUBLIC
--
--  PURPOSE:  Initializes instance data
--
--  INVOKED:  By Workplace Shell, upon instantiation of the object instance.
--

override wpModifyPopupMenu;
--
```

```
-- METHOD:   wpModifyPopupMenu                          PUBLIC
--
-- PURPOSE:  Adds an additional "Lock" item to the object's context menu.
--
-- INVOKED:  By Workplace Shell, upon instantiation of the object instance.
--

override wpMenuItemSelected;
--
-- METHOD:   wpMenuItemSelected                         PUBLIC
--
-- PURPOSE:  Processes the user's selections from the context menu.  The
--           overridden method processes only the added "Lock" item, before
--           invoking the parent's default processing to handle other items.
--
-- INVOKED:  By Workplace Shell, upon selection of a menu item by the user.
--

override wpOpen;
--
-- METHOD:   wpOpen                                     PUBLIC
--
-- PURPOSE:  Only allows a folder to be opened if the folder is unlocked, or
--           if the user supplies the correct password in response to the
--           dialog.
--
-- INVOKED:  By Workplace Shell, upon selection of the "Open" menu item by
--           the user.
--

override wpSetTitle;
--
-- METHOD:   wpSetTitle                                 PUBLIC
--
-- PURPOSE:  Sets the folder's title (icon text) to have the phrase <Locked>
--           as a suffix if the folder is locked, or removes this suffix if
--           the folder is unlocked.
--
-- INVOKED:  By wpOpen to set the unlocked state, and by LockFolder to set
--           the locked state.
--

override wpSetup;
--
-- METHOD:   wpSetup                                    PUBLIC
--
-- PURPOSE:  Sets folder properties based upon a setup string passed by the
--           object's creator as part of the WinCreateObject() call.  The
--           overridden method simply processes the PASSWORD keyword to set
--           the folder's password immediately upon instantiation, before
--           invoking the parent's default processing to handle all other
--           keywords.
--
-- INVOKED:  By the Workplace Shell, upon instantiation of the object
--           instance.
--

override wpSaveState;
--
-- METHOD:   wpSaveState                                PUBLIC
--
-- PURPOSE:  Saves the object instance's persistent state data.  The
--           overridden method simply saves the password data, then invokes
--           the parent's default processing to handle any other instance
--           data defined by ancestor classes.
--
-- INVOKED:  By the Workplace Shell, when the object becomes dormant.
--

override wpRestoreState;
--
-- METHOD:   wpRestoreState                             PUBLIC
--
-- PURPOSE:  Restores the object instance's persistent state data.  The
--           overridden method simply restores the password data, then
```

```
--          invokes the parent's default processing to handle any other
--          instance data defined by ancestor classes.
--
--  INVOKED:  By the Workplace Shell, when the object becomes awake.
--

override wpSetIcon;
--
--  METHOD:   wpSetIcon                                   PUBLIC
--
--  PURPOSE:  This class method returns the handle to the correct icon for
--            the object.
--
--  INVOKED:  -
--

override wpclsQueryTitle, classmethod;
--
--  METHOD:   wpclsQueryTitle                             PUBLIC
--
--  PURPOSE:  This class method returns the default folder title for any
--            instance of the password protected folder class.  This title
--            is used if a title is not supplied in the WinCreateObject()
--            call.
--
--  INVOKED:  By the Workplace Shell, upon instantiation of the object
--            instance.
--

override wpclsInitData, classmethod;
--
--  METHOD:   wpclsInitData                               PUBLIC
--
--  PURPOSE:  This class method allows the initialization of any class data
--            items.  The overridden method simply obtains a module handle
--            to be used when accessing Presentation Manager resources, then
--            invokes the parent's default processing.
--
--  INVOKED:  By the Workplace Shell, upon loading the class DLL.
--

override wpclsQueryIcon, classmethod;
--
--  METHOD:   wpclsQueryIcon                                PUBLIC
--
--  PURPOSE:  This class method returns the handle to the default icon for
--            the class.  This method is not used in the current version,
--            but could be used if different icons are to be used for the
--            locked and unlocked states.
--
--  INVOKED:  -
--

override wpclsUnInitData, classmethod;
--
--  METHOD:   wpclsUnInitData                             PUBLIC
--
--  PURPOSE:  This class method allows the release of any class data items
--            or resources.  The overridden method releases the module handle
--            obtained by wpclsInitData, then invokes the parent's default
--            processing.
--
--  INVOKED:  By the Workplace Shell, upon unloading the class DLL.
--
```

E.1.2 Source Code for the PWFolder.C file

```
/*************************************************************************/
/*                                                                       */
/* ITSC Redbook OS/2 v2.0 Sample Program                                 */
/*                                                                       */
/*                                                                       */
/* PWFOLDER.C                                                            */
/*                                                                       */
/*                                                                       */
/*************************************************************************/

/*
 * This file was generated by the SOM Compiler.
 * FileName: pwfolder.c.
 * Generated using:
 *     SOM Precompiler spc: 1.22
 *     SOM Emitter emitc: 1.24
 */
#define INCL_WIN
#define INCL_DOS
#define INCL_GPIBITMAPS
#define INCL_WPCLASS
#define INCL_WPFOLDER

/*****************************************************************************/
/* System-defined header files                                               */
/*****************************************************************************/
#include <os2.h>

#include <pmwp.h>  /* eventually will be #define INCL_WINWORKPLACE */

#include <string.h>
#include <stdio.h>
#include <memory.h>
#include <stdlib.h>

/*****************************************************************************/
/* Function prototype for dialog proc                                        */
/*****************************************************************************/
MRESULT EXPENTRY PasswordDlgProc(HWND hwndDlg,
                                 ULONG msg,
                                 MPARAM mp1,
                                 MPARAM mp2);

/*****************************************************************************/
/* Dialog definitions header file                                            */
/*****************************************************************************/
#include "dialog.h"

/*****************************************************************************/
/* Global data                                                               */
/*****************************************************************************/
HMODULE  hmodThisClass;                        /* Module handle            */
HPOINTER hLockedIcon;                          /* Handle for locked icon   */
HPOINTER hUnlockedIcon;                        /* Handle for unlocked icon */

PSZ      DefaultClassTitle = "Password Folder"; /* Default folder title     */


#define PWFolder_Class_Source
#include "pwfolder.ih"

/*
 *
 *   METHOD:   QueryInfo                                   PRIVATE
 *
 *   PURPOSE:  Copies instance data into the PWF_INFO structure.
 *
 *   INVOKED:  From PasswordDlgProc
 *
 */

SOM_Scope BOOL   SOMLINK pwfolder_QueryInfo(PWFolder *somSelf,
                 PPWF_INFO pPWFolderInfo)
{
    PWFolderData *somThis =                    /* Get instance data pointer */
```

```
        PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",             /* Set debug info           */
        "pwfolder_QueryInfo");

    strcpy(pPWFolderInfo->szCurrentPassword,    /* Set user-entered password */
        _szCurrentPassword);
    strcpy(pPWFolderInfo->szPassword,           /* Set folder password      */
        _szPassword);
    strcpy(pPWFolderInfo->szUserid,             /* Set userid               */
        _szUserid);

    return (BOOL) 0;                            /* Return                   */
}

/*
 *
 *   METHOD:   SetInfo                                    PRIVATE
 *
 *   PURPOSE:  Sets instance data from the PWF_INFO structure.
 *
 *   INVOKED:  From PasswordDlgProc
 *
 */

SOM_Scope BOOL   SOMLINK pwfolder_SetInfo(PWFolder *somSelf,
                 PPWF_INFO pPWFolderInfo)
{
    PWFolderData *somThis =                     /* Get instance data pointer */
        PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",             /* Set debug info           */
        "pwfolder_QueryInfo");

    strcpy(_szCurrentPassword,                  /* Save user-entered p'word  */
        pPWFolderInfo->szCurrentPassword);
    strcpy(_szPassword,                         /* Save folder password     */
        pPWFolderInfo->szPassword);
    strcpy(_szUserid,                           /* Save userid              */
        pPWFolderInfo->szUserid);

    return (BOOL) 0;                            /* Return                   */
}

/*
 *
 *   METHOD:   LockFolder                                 PUBLIC
 *
 *   PURPOSE:  Invalidates the current password, thereby locking the folder.
 *
 *   INVOKED:  From _wpMenuItemSelected
 *
 */

SOM_Scope BOOL   SOMLINK pwfolder_LockFolder(PWFolder *somSelf)
{
    BOOL bSuccess;

    PWFolderData *somThis =                     /* Get instance data pointer */
        PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",             /* Set debug info           */
        "pwfolder_QueryInfo");

    strcpy(_szCurrentPassword,"NOPASSWD");      /* Invalid user-entered     */
                                                /* password                 */
    _wpSetTitle(somSelf,                        /* Set folder title to      */
                _wpQueryTitle(somSelf) );       /* locked state             */

    bSuccess=_wpSetIcon(somSelf,                /* Set icon to locked state */
                        hLockedIcon);

    _wpSaveImmediate(somSelf);                  /* Rember this state        */
    return (BOOL) 0;                            /* Return                   */
}

/*
 *
```

```
 *   METHOD:   wpInitData                                PUBLIC
 *
 *   PURPOSE:  Initializes instance data
 *
 *   INVOKED:  By Workplace Shell, upon instantiation of the object instance.
 *
 */

SOM_Scope void   SOMLINK pwfolder_wpInitData(PWFolder *somSelf)
{
    CHAR  ErrorBuffer[100];

    PWFolderData *somThis =                     /* Get instance data pointer */
                 PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",             /* Set debug info            */
                        "pwfolder_wpInitData");

    parent_wpInitData(somSelf);                 /* Invoke default processing */

    strcpy(_szCurrentPassword,"password");      /* Initialise folder in the  */
    strcpy(_szPassword,"password");             /* unlocked state            */
}

/*
 *
 *   METHOD:   wpModifyPopupMenu                         PUBLIC
 *
 *   PURPOSE:  Adds an additional "Lock" item to the object's context menu.
 *
 *   INVOKED:  By Workplace Shell, upon instantiation of the object instance.
 *
 */

SOM_Scope BOOL   SOMLINK pwfolder_wpModifyPopupMenu(PWFolder *somSelf,
                 HWND hwndMenu,
                 HWND hwndCnr,
                 ULONG iPosition)
{
    PWFolderData *somThis =                     /* Get instance data pointer */
                 PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",             /* Set debug info            */
                        "pwfolder_wpModifyPopupMenu");

    _wpInsertPopupMenuItems(somSelf,            /* Insert menu item          */
                            hwndMenu,           /* Menu handle               */
                            iPosition,          /* Default position          */
                            hmodThisClass,      /* Module handle             */
                            ID_CXTMENU_LOCK,    /* Menu item identifier      */
                            0);                 /* No submenu identifier     */

    return(parent_wpModifyPopupMenu(somSelf,    /* Invoke default processing */
                                    hwndMenu,
                                    hwndCnr,
                                    iPosition));
}

/*
 *
 *   METHOD:   wpMenuItemSelected                        PUBLIC
 *
 *   PURPOSE:  Processes the user's selections from the context menu.  The
 *             overridden method processes only the added "Lock" item, before
 *             invoking the parent's default processing to handle other items.
 *
 *   INVOKED:  By Workplace Shell, upon selection of a menu item by the user.
 *
 */

SOM_Scope BOOL   SOMLINK pwfolder_wpMenuItemSelected(PWFolder *somSelf,
                 HWND hwndFrame,
                 ULONG ulMenuId)
{
    PWFolderData *somThis =                     /* Get instance data pointer */
                 PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",             /* Set debug info            */
```

```
                        "pwfolder_wpMenuItemSelected");

    switch( ulMenuId )                          /* Switch on item identifier */
    {
       case IDM_LOCK:                           /* Lock item selected        */
          _LockFolder(somSelf);                 /* Invoke _LockFolder method */
          break;

       default:                                 /* All other items           */
          parent_wpMenuItemSelected(somSelf,    /* Invoke default processing */
                                    hwndFrame,
                                    ulMenuId);
          break;
    }
}

/*
 *
 *   METHOD:   wpOpen                                        PUBLIC
 *
 *   PURPOSE:  Only allows a folder to be opened if the folder is unlocked, or
 *             if the user supplies the correct password in response to the
 *             dialog.
 *
 *   INVOKED:  By Workplace Shell, upon selection of the "Open" menu item by
 *             the user.
 *
 */

SOM_Scope HWND   SOMLINK pwfolder_wpOpen(PWFolder *somSelf,
                 HWND hwndCnr,
                 ULONG ulView,
                 ULONG param)
{
    ULONG    ulResult;                          /* Return value              */
    CHAR     szTitle[100];                      /* Folder title buffer       */
    PVOID    pCreateParam;

    PWFolderData *somThis =                     /* Get instance data pointer */
                 PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",             /* Set debug info            */
                        "pwfolder_wpOpen");

    if ((strcmp(_szCurrentPassword,             /* If folder is not locked   */
         _szPassword)) == 0)
    {
        return(parent_wpOpen(somSelf,           /* Allow open to proceed in  */
                             hwndCnr,           /* normal way using default  */
                             ulView,            /* processing                */
                             param));
    }
    pCreateParam = malloc( sizeof(ULONG) );     /* Allocate memory to pass a */
                                                /* ULONG to the dialog proc  */
    *((PULONG)pCreateParam) = (ULONG)somSelf;   /* Put the somSelf pointer   */
                                                /* in the CreateParam memory */

    ulResult = WinDlgBox(HWND_DESKTOP,          /* Display password dialog   */
                         HWND_DESKTOP,          /* Desktop is owner          */
                         PasswordDlgProc,       /* Dialog procedure address  */
                         hmodThisClass,         /* Module handle             */
                         ID_DLG_PASSWORD,       /* Dialog resource id        */
                         pCreateParam );        /* Create Param holding the  */
                                                /* pointer to this object    */

    if (ulResult == DID_OK )                    /* If user hit OK button     */
       {
       if ((strcmp(_szCurrentPassword,          /* If password is correct    */
                   _szPassword)) == 0)
          {
          strcpy(szTitle,                       /* Get title string          */
                 _wpQueryTitle(somSelf));
          szTitle[strlen(szTitle)-9] = '\0';    /* Remove <LOCKED>           */
          _wpSetTitle(somSelf,szTitle);         /* Reset title string        */

          _wpSetIcon(somSelf,                   /* Set icon to unlocked      */
```

```
                    hUnlockedIcon);             /* state                    */

         return(parent_wpOpen(somSelf,          /* Allow open to proceed in  */
                              hwndCnr,          /* normal way using default  */
                              ulView,           /* processing                */
                              param));
         }
      else                                      /* Password is incorrect     */
         {
         WinMessageBox(HWND_DESKTOP,            /* Display message to user   */
                       HWND_DESKTOP,
                       "Password incorrect. Folder remains locked.",
                       "Password Failed",
                       0,
                       MB_OK |
                       MB_CUAWARNING );
         return((HWND)0);                       /* Return NULL handle        */
         }
      }
}

/*
 *
 *   METHOD:   wpSetTitle                                   PUBLIC
 *
 *   PURPOSE:  Sets the folder's title (icon text) to have the phrase <Locked>
 *             as a suffix if the folder is locked, or removes this suffix if
 *             the folder is unlocked.
 *
 *   INVOKED:  By wpOpen to set the unlocked state, and by LockFolder to set
 *             the locked state.
 *
 */

SOM_Scope BOOL   SOMLINK pwfolder_wpSetTitle(PWFolder *somSelf,
                 PSZ pszNewTitle)
{
   CHAR szBuf[100];                             /* Character buffer          */

   PWFolderData *somThis =                      /* Get instance data pointer */
                 PWFolderGetData(somSelf);
   PWFolderMethodDebug("PWFolder",              /* Set debug info            */
                       "pwfolder_wpSetTitle");

   strcpy(szBuf,pszNewTitle);                   /* Get current title         */

   if ((strcmp(_szCurrentPassword,              /* If folder is locked       */
               _szPassword)) != 0)
   {
      if ((strstr(szBuf,"LOCKED")) == NULL)     /* and <LOCKED> not in       */
      {                                         /* current title             */
         strcat(szBuf," <LOCKED>");             /* Add <LOCKED> to title     */
      }
   }
   return (parent_wpSetTitle(somSelf,szBuf));   /* Invoke default processing */
}

/*
 *
 *   METHOD:   wpSetup                                      PUBLIC
 *
 *   PURPOSE:  Sets folder properties based upon a setup string passed by the
 *             object's creator as part of the WinCreateObject() call.  The
 *             overridden method simply processes the PASSWORD keyword to set
 *             the folder's password immediately upon instantiation, before
 *             invoking the parent's default processing to handle all other
 *             keywords.
 *
 *   INVOKED:  By the Workplace Shell, upon instantiation of the object
 *             instance.
 *
 */

SOM_Scope BOOL   SOMLINK pwfolder_wpSetup(PWFolder *somSelf,
```

```
                PSZ pszSetupString)
{
    CHAR  pszInitPword[20];                     /* Character buffer          */
    BOOL  bFound;                               /* Success flag              */
    ULONG Length;                               /* Returned length           */

    PWFolderData *somThis =                     /* Get instance data pointer */
                 PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",             /* Get debug info            */
                        "pwfolder_wpSetup");

    if (*pszSetupString != '\0')                /* If string is present      */
       {
       bFound=_wpScanSetupString(somSelf,       /* Parse setup string to     */
                                 pszSetupString, /* find PASSWORD keyword    */
                                 "PASSWORD",
                                 pszInitPword,  /* Buffer for keyword value  */
                                 &Length);      /* Length of returned string */

       if (bFound)
          {
          strcpy(_szPassword,                   /* Initialize folder         */
                 pszInitPword);                 /* password                  */
          strcpy(_szCurrentPassword,            /* Initialize user-entered   */
                 pszInitPword);                 /* password                  */
          }
       }
    return(parent_wpSetup(somSelf,              /* Invoke default processing */
                          pszSetupString));
}

/*
 *
 *   METHOD:   wpSaveState                                    PUBLIC
 *
 *   PURPOSE:  Saves the object instance's persistent state data.  The
 *             overridden method simply saves the password data, then invokes
 *             the parent's default processing to handle any other instance
 *             data defined by ancestor classes.
 *
 *   INVOKED:  By the Workplace Shell, when the object becomes dormant.
 *
 */

SOM_Scope BOOL   SOMLINK pwfolder_wpSaveState(PWFolder *somSelf)
{
    PWFolderData *somThis =                     /* Get instance data pointer */
                 PWFolderGetData(somSelf);
    PWFolderMethodDebug("PWFolder",             /* Set debug info            */
                        "pwfolder_wpSaveState");

    _wpSaveString(somSelf,                      /* Save folder password      */
                  "PWFolder",                   /* Class name                */
                  1L,                           /* Class-defined key         */
                  _szPassword);                 /* String to be saved        */
    _wpSaveString(somSelf,                      /* Save user-entered p'word  */
                  "PWFolder",                   /* Class name                */
                  2L,                           /* Class-defined key         */
                  _szCurrentPassword);          /* String to be saved        */

    return(parent_wpSaveState(somSelf));        /* Invoke default processing */
}

/*
 *
 *   METHOD:   wpRestoreState                                 PUBLIC
 *
 *   PURPOSE:  Restores the object instance's persistent state data.  The
 *             overridden method simply restores the password data, then
 *             invokes the parent's default processing to handle any other
 *             instance data defined by ancestor classes.
 *
 *   INVOKED:  By the Workplace Shell, when the object becomes awake.
 *
 */
```

```
SOM_Scope BOOL   SOMLINK pwfolder_wpRestoreState(PWFolder *somSelf,
                ULONG ulReserved)
{
   ULONG ulRetLength;                          /* Length of returned string */

   PWFolderData *somThis =                     /* Get instance data pointer */
                PWFolderGetData(somSelf);
   PWFolderMethodDebug("PWFolder",             /* Set debug info            */
                       "pwfolder_wpRestoreState");

   _wpRestoreString(somSelf,                   /* Restore folder password   */
                    "PWFolder",                /* Class name                */
                    1L,                        /* Class-defined key         */
                    _szPassword,               /* String to be restored     */
                    &ulRetLength);             /* Length of returned string */
   _wpRestoreString(somSelf,                   /* Restore user-entered pwd  */
                    "PWFolder",                /* Class name                */
                    2L,                        /* Class-defined key         */
                    _szCurrentPassword,        /* String to be restored     */
                    &ulRetLength);             /* Length of returned string */

   if ((strcmp(_szCurrentPassword,             /* If folder is locked       */
               _szPassword)) != 0)
   {
      _wpSetIcon(somSelf, hLockedIcon);        /* Set icon to locked state  */
   }

   return(parent_wpRestoreState(somSelf,       /* Invoke default processing */
                                ulReserved));
}

/*
 *
 *   METHOD:   wpSetIcon                                   PUBLIC
 *
 *   PURPOSE:  This class method returns the handle to the correct icon for
 *             the object.
 *
 *   INVOKED:
 *
 */

SOM_Scope BOOL   SOMLINK pwfolder_wpSetIcon(PWFolder *somSelf,
                HPOINTER hptrNewIcon)
{
   PWFolderData *somThis = PWFolderGetData(somSelf);
   PWFolderMethodDebug("PWFolder","pwfolder_wpSetIcon");

   if ((strcmp(_szCurrentPassword,         /* If password is correct     */
               _szPassword)) == 0)
   {
      return (parent_wpSetIcon(somSelf,
                    hUnlockedIcon));       /* return pointer to unlocked */
   }
   else                                    /* otherwise                  */
   {
      return (parent_wpSetIcon(somSelf,
                    hLockedIcon));         /* return locked icon pointer */
   }
}

#undef SOM_CurrentClass

#define SOM_CurrentClass SOMMeta
/*
 *
 *   METHOD:   wpclsQueryTitle                             PUBLIC
 *
 *   PURPOSE:  This class method returns the default folder title for any
 *             instance of the password protected folder class.  This title
 *             is used if a title is not supplied in the WinCreateObject()
 *             call.
 *
 *   INVOKED:  By the Workplace Shell, upon instantiation of the object
```

```
 *            instance.
 *
 */

SOM_Scope PSZ   SOMLINK pwfoldercls_wpclsQueryTitle(M_PWFolder *somSelf)
{
    /* M_PWFolderData *somThis = M_PWFolderGetData(somSelf); */

    M_PWFolderMethodDebug("M_PWFolder",          /* Set debug info           */
                          "pwfoldercls_wpclsQueryTitle");

    return(DefaultClassTitle);                   /* Return default title     */
}

/*
 *
 *   METHOD:   wpclsQueryIcon                                PUBLIC
 *
 *   PURPOSE:  This class method returns the handle to the default icon for
 *             the class.  This method is not used in the current version,
 *             but could be used if different icons are to be used for the
 *             locked and unlocked states.
 *
 *   INVOKED:
 *
 */

SOM_Scope HPOINTER   SOMLINK pwfoldercls_wpclsQueryIcon(M_PWFolder *somSelf)
{
    /* M_PWFolderData *somThis = M_PWFolderGetData(somSelf); */
    PWFolderData *somThis =                      /* Get instance data pointer */
                PWFolderGetData(somSelf);

    M_PWFolderMethodDebug("M_PWFolder",          /* Set debug info           */
                          "pwfoldercls_wpclsQueryIcon");

    return(hUnlockedIcon);
}

/*
 *
 *   METHOD:   wpclsInitData                                 PUBLIC
 *
 *   PURPOSE:  This class method allows the initialization of any class data
 *             items.  The overridden method simply obtains a module handle
 *             to be used when accessing Presentation Manager resources, then
 *             invokes the parent's default processing.
 *
 *   INVOKED:  By the Workplace Shell, upon loading the class DLL.
 *
 */

SOM_Scope void   SOMLINK pwfoldercls_wpclsInitData(M_PWFolder *somSelf)
{
    CHAR  ErrorBuffer[100];                      /* Error buffer             */

    /* M_PWFolderData *somThis = M_PWFolderGetData(somSelf); */

    M_PWFolderMethodDebug("M_PWFolder",          /* Set debug info           */
                          "pwfoldercls_wpclsInitData");

    DosLoadModule((PSZ) ErrorBuffer,             /* Obtain DLL module handle */
                  sizeof(ErrorBuffer),
                  "PWFOLDER",                    /* Module name              */
                  &hmodThisClass);                       /* Module handle          */

    hLockedIcon=WinLoadPointer(HWND_DESKTOP,     /* Load icons               */
                               hmodThisClass,
                               ID_LOCK);
    hUnlockedIcon=WinLoadPointer(HWND_DESKTOP,
                                 hmodThisClass,
                                 ID_UNLOCK);

    parent_wpclsInitData(somSelf);               /* Invoke default processing */
}
```

```
/*
 *
 *   METHOD:   wpclsUnInitData                                PUBLIC
 *
 *   PURPOSE:  This class method allows the release of any class data items
 *             or resources.  The overridden method releases the module handle
 *             obtained by wpclsInitData, then invokes the parent's default
 *             processing.
 *
 *   INVOKED:  By the Workplace Shell, upon unloading the class DLL.
 *
 */

SOM_Scope void   SOMLINK pwfoldercls_wpclsUnInitData(M_PWFolder *somSelf)
{
   /* M_PWFolderData *somThis = M_PWFolderGetData(somSelf); */

   M_PWFolderMethodDebug("M_PWFolder",          /* Set debug info          */
                         "pwfoldercls_wpclsUnInitData");

   WinDestroyPointer(hLockedIcon);              /* Free icons              */
   WinDestroyPointer(hUnlockedIcon);

   DosFreeModule(hmodThisClass);                        /* Free module handle       */

   parent_wpclsUnInitData(somSelf);
}

/*****************************************************************************/
/*                                                                           */
/* PROCEDURE NAME:  PasswordDlgProc                                          */
/*                                                                           */
/* description:  Dialog procedure for password dialog                        */
/*                                                                           */
/* invoked:  By Presentation Manager, in response to folder issuing          */
/*           WinDlgBox() call from _wpOpen method.                           */
/*                                                                           */
/*****************************************************************************/
MRESULT EXPENTRY PasswordDlgProc(HWND hwndDlg,
                                 ULONG msg,
                                 MPARAM mp1,
                                 MPARAM mp2)
{
   PWFolder *aPWF;                              /* Define SOM pointer       */
   PWF_INFO pwfolderInfo;                       /* Define password structure */
   CHAR     szTemp[100];                        /* Character buffer         */

   switch (msg)                                 /* Determine message class  */
   {
      case WM_INITDLG:                          /* Dialog being initialized */

         WinSetWindowULong(hwndDlg,             /* Store SOM pointer in     */
                           QWL_USER,            /* window word QWL_USER     */
                           *((PULONG)mp2));
         free(mp2);                             /* Free Create Param memory */
         break;

      case WM_COMMAND:                          /* User hit a pushbutton    */

         aPWF = (PWFolder *)WinQueryWindowULong(hwndDlg, QWL_USER);

         _QueryInfo(aPWF,                       /* Get password data        */
                    &pwfolderInfo);

         switch (SHORT1FROMMP(mp1))             /* Which button was hit?    */
         {
            case DID_OK:                        /* Okay button              */

               WinQueryDlgItemText(hwndDlg,     /* Get text from entryfield */
                                   ID_EF_PASSWORD,
                                   sizeof(szTemp),
                                   (PSZ)szTemp);

                                                /* Copy to password data    */
```

```
                strcpy(pwfolderInfo.szCurrentPassword, szTemp);

                _SetInfo(aPWF,&pwfolderInfo);      /* Set instance data        */
                WinDismissDlg(hwndDlg,DID_OK);     /* Dismiss dialog           */
                break;

            case DID_CANCEL:                       /* Cancel button hit        */

                WinDismissDlg(hwndDlg,             /* Dismiss dialog           */
                              DID_CANCEL);
                break;
        }
        return(MRESULT)TRUE;                       /* Return from WM_COMMAND   */
        break;
    }
    return(WinDefDlgProc(hwndDlg,                  /* Invoke default PM message */
                         msg,                      /* handling                 */
                         mp1,
                         mp2));
}
```

E.1.3 Source Code for the PWFolder.MAK file

```
#*****************************************************************************
#  Dot directive definition area (usually just suffixes)
#*****************************************************************************

.SUFFIXES: .c .obj .dll .csc .sc .h .ih .ph .psc .rc .res

#*****************************************************************************
#  Environment Setup for the component(s).
#*****************************************************************************

SOMTEMP = .\somtemp
SCPATH  = D:\toolkt20\sc
HPATH   = D:\toolkt20\c\os2h
LIBPATH = D:\toolkt20\os2lib

!if [set SMINCLUDE=.;$(SCPATH);] || \
    [set SMTMP=$(SOMTEMP)] || \
    [set SMEMIT=ih;h;ph;psc;sc;c;def]
!endif

!if [cd $(SOMTEMP)]
!  if [md $(SOMTEMP)]
!    error Error creating $(SOMTEMP) directory
!  endif
!else
!  if [cd ..]
!    error Error could not cd .. from $(SOMTEMP) directory
!  endif
!endif

#
# Compiler/tools Macros
#

CC         = icc /c /Ge- /Gd- /Se /Re /ss /Ms /Gm+
LINK       = link386
LDFLAGS    = /noi /map /nol /nod /exepack /packcode /packdata /align:16 /information
#LIBS       = som.lib os2386.lib dde4sbs.lib dde4nbs.lib
LIBS       = som.lib os2386.lib dde4nbs.lib

#*****************************************************************************
# Set up Macros that will contain all the different dependencies for the
# executables and dlls etc. that are generated.
#*****************************************************************************

OBJS       = pwfolder.obj

#*****************************************************************************
#   Setup the inference rules for compiling source code to
#   object code.
#*****************************************************************************
```

```
.c.obj:
        $(CC) -I$(HPATH) -c $<

.csc.c:
        sc  $<

all: pwfolder.dll

pwfolder.obj: $*.c $*.ih $*.h  $*.sc

pwfolder.dll: $(OBJS) pwfolder.res
         $(LINK) $(LDFLAGS) $(OBJS),$@,,$(LIBS),$*;
         rc $*.res $*.dll
         mapsym pwfolder.map

pwfolder.res: pwfolder.rc
         rc -r $*.rc $*.res
```

E.1.4 Source Code for the PWFolder.RC file

```
#define INCL_WIN
#include <os2.h>

#include "dialog.h"

ICON  ID_LOCK     LOCKED.ICO
ICON  ID_UNLOCK   UNLOCKED.ICO

MENU ID_CXTMENU_LOCK LOADONCALL MOVEABLE DISCARDABLE
BEGIN
   MENUITEM " Lock Folder", IDM_LOCK
END


DLGTEMPLATE ID_DLG_PASSWORD LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG  "Password", ID_DLG_PASSWORD, 35, 26, 224, 76, FS_SYSMODAL |
            FS_SCREENALIGN | WS_VISIBLE, FCF_TITLEBAR
    BEGIN
        ENTRYFIELD      "", ID_EF_PASSWORD, 13, 38, 97, 8, NOT ES_AUTOSCROLL |
                        ES_MARGIN | ES_UNREADABLE
        LTEXT           "Folder is locked.  Please enter password to open",
                        102, 10, 56, 197, 8
        DEFPUSHBUTTON   "Ok", DID_OK, 36, 9, 40, 14
        PUSHBUTTON      "Cancel", DID_CANCEL, 137, 10, 40, 14
    END
END
```

E.1.5 Source Code for the DIALOG.H file

```
 xmpcod1.
#define ID_CXTMENU_LOCK            0x6501
#define IDM_LOCK                   0x6502
#define IDM_QUERY                  0x6503

#define ID_DLG_PASSWORD            100
#define ID_EF_PASSWORD             103

#define ID_DLG_FIND                200
#define ID_EF_TELNUMBER            201
#define ID_EF_SURNAME              203

#define ID_LOCK                    300
#define ID_UNLOCK                  301
```

E.2 Source Code for the PWFinanceFile Object

This section list the source code necessary to generate the dynamic link library (DLL) file PWFIN.DLL.

E.2.1 Source Code for the PWFin.CSC file

```
##############################################################
#                                                            #
#  pwFin.CSC          (c) IBM Corporation 1992               #
#                                                            #
#  This class derives from WPDataFile, and is used to        #
#  represent a FinanceFile which is protected by a password. #
#                                                            #
##############################################################

#
# Include the class definition file for the parent class
#
include <wpdataf.sc>
#
#   Define the new class
#
class: PWFinanceFile,
       file stem = pwFin,
       external prefix = pwFinanceFile_,
       class prefix = pwFinanceFilecls_,
       major version = 1,
       minor version = 1,
       local;

-- PWFinanceFile is a Password protected FinanceFile.
-- It is derived as follows:
--        SOMObject
--          - WPObject
--             - WPFileSystem
--                - WPDataFile
--                   - PWFinanceFile

#
# Specify the parent class
#
parent: WPDataFile;

#
# Specify the release order of new methods
#
release order: LockFinanceFile;

#
# Passthru a debug message box to the .ih file
# (for inclusion in the .c file)
#
passthru: C.ih, after;
#define INCL_PM
#define INCL_BSE
#define INCL_DOSERRORS
#include <wppgm.h>
#include <wppgmf.h>
#include <stdio.h>
#include <os2.h>

// force SOM to output all debug information to the Communications Port 1

int myReplacementForSOMOutChar (char c)
{
static FILE *fdebug = NULL;

if (!fdebug) {
  fdebug = fopen("COM1","w");

  if (!fdebug) return 0;     /* failed to open COM1: */
```

```
  }
fputc(c,fdebug);
fflush(fdebug);
/*if (c=='\n') fflush(fdebug);*/
return 1;
}

/*
 *   The following structures will be used to store window specific data
 *   and a pointer to the object that created the window/dialog.
 *
 *   They're allocated when the Client window is created.  This
 *   allows us to pass the *somSelf pointer and use it in our
 *   window and dialog procedures (the system only passes this
 *   pointer to methods).
 */
   typedef struct _WINDOWDATA
   {
      USHORT   cb;            /* size of this structure */
PWFinanceFile *somSelf;       /* pointer to this instance */
      USEITEM  UseItem;       /* global class usage information */
      VIEWITEM ViewItem;      /* global class view information */
      LONG     x;             /* x position of car in open view */
      LONG     y;             /* y position of car in open view */
      LONG     xDir;          /* x direction */
      LONG     yDir;          /* y direction */
   } WINDOWDATA;
   typedef WINDOWDATA *PWINDOWDATA;

endpassthru;

#
# Passthru private definitions to the .ph file
# (for inclusion in the .c file)
#
passthru: C.ph;

typedef struct _PWF_INFO {                        /* Define password structure */
  CHAR   szPassword[20];              /* FinanceFile current password   */
  CHAR   szCurrentPassword[20];       /* User-entered password     */
  CHAR   szUserid[20];                /* Userid                    */
} PWF_INFO;
typedef PWF_INFO *PPWF_INFO;                      /* Define pointer type       */

endpassthru;

#
# Define instance data for the class
#
data:
CHAR szPassword[20];
--  This is the password which locks the FinanceFile

CHAR szCurrentPassword[20];
--  This is the password the user has typed in to be
--  checked against the lock password

CHAR szUserid[20];
--  Userid


#
# Define new methods
#
methods:

BOOL QueryInfo(PPWF_INFO pPWFinanceFileInfo), private;
--
--  METHOD:   QueryInfo                                    PRIVATE
--
--  PURPOSE:  Copies instance data into the PWF_INFO structure.
--
--  INVOKED:  From PasswordDlgProc
--
```

```
BOOL SetInfo(PPWF_INFO pPWFinanceFileInfo), private;
--
--  METHOD:   SetInfo                                    PRIVATE
--
--  PURPOSE:  Sets instance data from the PWF_INFO structure.
--
--  INVOKED:  From PasswordDlgProc
--

BOOL LockFinanceFile();
--
--  METHOD:   LockFinanceFile                                PUBLIC
--
--  PURPOSE:  Invalidates the current password, thereby locking the FinanceFile.
--
--  INVOKED:  From _wpMenuItemSelected
--

#
# Specify methods being overridden
#
override wpFilterPopupMenu;
--
--  METHOD:   wpFilterPopupMenu                          PUBLIC
--
--  PURPOSE:  This class method is called when the user asks for the context
--            (popup) menu.
--
--  INVOKED:  -
--

override wpPrintObject;
--
--  METHOD:   wpPrintObject                              PUBLIC
--
--  PURPOSE:  This class method allows this object to format it's output for
--            printing.
--
--  INVOKED:  -
--

override wpDraggedOverObject;
--
--  METHOD:   wpDraggedOverObject                        PUBLIC
--
--  PURPOSE:  Checks to see that the file is unlocked if user wants to
--            drop the object on a program.
--
--  INVOKED:  -
--

override wpDragOver;
--
--  METHOD:   wpDragOver                                 PUBLIC
--
--  PURPOSE:  Checks to see that the object being dragged over me is also
--            either of my class or derived from me.
--
--  INVOKED:  By Workplace Shell, when an object is being dragged over this
--            object.
--

override wpDrop;
--
--  METHOD:   wpDrop                                     PUBLIC
--
--  PURPOSE:  To receive a dropped object.
--
--  INVOKED:  By Workplace Shell, when another object has been dropped on
--            this object.
--

override wpInitData;
--
--  METHOD:   wpInitData                                 PUBLIC
```

```
--
--  PURPOSE:  Initializes instance data
--
--  INVOKED:  By Workplace Shell, upon instantiation of the object instance.
--

override wpModifyPopupMenu;
--
--  METHOD:   wpModifyPopupMenu                          PUBLIC
--
--  PURPOSE:  Adds an additional "Lock" item to the object's context menu.
--
--  INVOKED:  By Workplace Shell, upon instantiation of the object instance.
--

override wpMenuItemSelected;
--
--  METHOD:   wpMenuItemSelected                         PUBLIC
--
--  PURPOSE:  Processes the user's selections from the context menu.  The
--            overridden method processes only the added "Lock" item, before
--            invoking the parent's default processing to handle other items.
--
--  INVOKED:  By Workplace Shell, upon selection of a menu item by the user.
--

override wpOpen;
--
--  METHOD:   wpOpen                                     PUBLIC
--
--  PURPOSE:  Only allows a FinanceFile to be opened if the FinanceFile is unlocked, or
--            if the user supplies the correct password in response to the
--            dialog.
--
--  INVOKED:  By Workplace Shell, from the parent wpViewObject method
--

override wpSetTitle;
--
--  METHOD:   wpSetTitle                                 PUBLIC
--
--  PURPOSE:  Sets the FinanceFile's title (icon text) to have the phrase <Locked>
--            as a suffix if the FinanceFile is locked, or removes this suffix if
--            the FinanceFile is unlocked.
--
--  INVOKED:  By wpOpen to set the unlocked state, and by LockFinanceFile to set
--            the locked state.
--

override wpSetup;
--
--  METHOD:   wpSetup                                    PUBLIC
--
--  PURPOSE:  Sets FinanceFile properties based upon a setup string passed by the
--            object's creator as part of the WinCreateObject() call.  The
--            overridden method simply processes the PASSWORD keyword to set
--            the FinanceFile's password immediately upon instantiation, before
--            invoking the parent's default processing to handle all other
--            keywords.
--
--  INVOKED:  By the Workplace Shell, upon instantiation of the object
--            instance.
--

override wpSaveState;
--
--  METHOD:   wpSaveState                                PUBLIC
--
--  PURPOSE:  Saves the object instance's persistent state data.  The
--            overridden method simply saves the password data, then invokes
--            the parent's default processing to handle any other instance
--            data defined by ancestor classes.
--
--  INVOKED:  By the Workplace Shell, when the object becomes dormant.
--
```

```
override wpRestoreState;
--
--  METHOD:   wpRestoreState                                PUBLIC
--
--  PURPOSE:  Restores the object instance's persistent state data.  The
--            overridden method simply restores the password data, then
--            invokes the parent's default processing to handle any other
--            instance data defined by ancestor classes.
--
--  INVOKED:  By the Workplace Shell, when the object becomes awake.
--

override wpSetIcon;
--
--  METHOD:   wpSetIcon                                     PUBLIC
--
--  PURPOSE:  This class method returns the handle to the correct icon for
--            the object.
--
--  INVOKED:  -
--

override wpAddFileTypePage;
--
--  METHOD:   wpAddFileTypePage                             PUBLIC
--
--  PURPOSE:  This class method replaces the type page with the it's own one
--            that only allows the FinanceFile Types.
--
--  INVOKED:  By the Workplace Shell, upon unloading the class DLL.
--

override wpclsQueryTitle, classmethod;
--
--  METHOD:   wpclsQueryTitle                               PUBLIC
--
--  PURPOSE:  This class method returns the default FinanceFile title for any
--            instance of the password protected FinanceFile class.  This title
--            is used if a title is not supplied in the WinCreateObject()
--            call.
--
--  INVOKED:  By the Workplace Shell, upon instantiation of the object
--            instance.
--

override wpclsInitData, classmethod;
--
--  METHOD:   wpclsInitData                                 PUBLIC
--
--  PURPOSE:  This class method allows the initialization of any class data
--            items.  The overridden method simply obtains a module handle
--            to be used when accessing Presentation Manager resources, then
--            invokes the parent's default processing.
--
--  INVOKED:  By the Workplace Shell, upon loading the class DLL.
--

override wpclsQueryIcon, classmethod;
--
--  METHOD:   wpclsQueryIcon                                PUBLIC
--
--  PURPOSE:  This class method returns the handle to the default icon for
--            the class.  This method is not used in the current version,
--            but could be used if different icons are to be used for the
--            locked and unlocked states.
--
--  INVOKED:  -
--

override wpclsUnInitData, classmethod;
--
--  METHOD:   wpclsUnInitData                               PUBLIC
--
--  PURPOSE:  This class method allows the release of any class data items
```

```
--          or resources.  The overridden method releases the module handle
--          obtained by wpclsInitData, then invokes the parent's default
--          processing.
--
--  INVOKED:  By the Workplace Shell, upon unloading the class DLL.
--
```

E.2.2 Source Code for the PWFin.C file

```
/*************************************************************************/
/*                                                                       */
/* ITSC Redbook OS/2 v2.0 Sample Program                                 */
/*                                                                       */
/*                                                                       */
/* PWFinanceFile.C                                                       */
/*                                                                       */
/*                                                                       */
/*************************************************************************/

/*
 * This file was generated by the SOM Compiler.
 * FileName: pwFinanceFile.c.
 * Generated using:
 *     SOM Precompiler spc: 1.22
 *     SOM Emitter emitc: 1.24
 */
#define INCL_WIN
#define INCL_DOS
#define INCL_GPIBITMAPS
#define INCL_WPCLASS
#define INCL_WPFOLDER
#define INCL_WINWORKPLACE
#define INCL_DOSERRORS

/*************************************************************************/
/* System-defined header files                                           */
/*************************************************************************/
#include <os2.h>

#include <pmwp.h>  /* eventually will be #define INCL_WINWORKPLACE */

#include <string.h>
#include <stdio.h>
#include <memory.h>
#include <stdlib.h>

/*************************************************************************/
/* Function prototype for dialog proc                                    */
/*************************************************************************/
MRESULT EXPENTRY PasswordDlgProc(HWND hwndDlg,
                                 ULONG msg,
                                 MPARAM mp1,
                                 MPARAM mp2);

/*************************************************************************/
/* Dialog definitions header file                                        */
/*************************************************************************/
#include "dialog.h"

/*************************************************************************/
/* Global data                                                           */
/*************************************************************************/
HPOINTER hLockedIcon;                        /* Handle for locked icon   */
HPOINTER hUnlockedIcon;                      /* Handle for unlocked icon */

HMODULE  hmodThisClass;

PSZ      DefaultClassTitle = "Password FinanceFile";  /* Default FinanceFile title     */

CHAR       szFinanceFileWindowClass[] = "Finance File Sample";
UCHAR      szFinanceFileClassTitle[CCHMAXPATH] = "";
```

```
#define PWFinanceFile_Class_Source
#include "pwFin.ih"

/*****************************************************************************/
/* Non Method Function Prototypes                                            */
/*****************************************************************************/
HWND PWFinanceFileInit (PWFinanceFile*);

MRESULT EXPENTRY FinanceFileWndProc ( HWND hwnd, ULONG msg, MPARAM mp1, MPARAM mp2 );

#define OBJECT_FROM_DRAGITEM(di) (di&&di->ulItemID ? OBJECT_FROM_PREC(di->ulItemID) : NULL)

/*
 *
 *   METHOD:   QueryInfo                                    PRIVATE
 *
 *   PURPOSE:  Copies instance data into the PWF_INFO structure.
 *
 *   INVOKED:  From PasswordDlgProc
 *
 */

SOM_Scope BOOL   SOMLINK pwFinanceFile_QueryInfo(PWFinanceFile *somSelf,
                 PPWF_INFO pPWFinanceFileInfo)
{
    PWFinanceFileData *somThis =                    /* Get instance data pointer */
          PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile",             /* Set debug info          */
          "pwFinanceFile_QueryInfo");

    strcpy(pPWFinanceFileInfo->szCurrentPassword,   /* Set user-entered password */
          _szCurrentPassword);
    strcpy(pPWFinanceFileInfo->szPassword,          /* Set FinanceFile password      */
          _szPassword);
    strcpy(pPWFinanceFileInfo->szUserid,            /* Set userid              */
          _szUserid);

    return (BOOL) 0;                            /* Return                  */
}

/*
 *
 *   METHOD:   SetInfo                                      PRIVATE
 *
 *   PURPOSE:  Sets instance data from the PWF_INFO structure.
 *
 *   INVOKED:  From PasswordDlgProc
 *
 */

SOM_Scope BOOL   SOMLINK pwFinanceFile_SetInfo(PWFinanceFile *somSelf,
                 PPWF_INFO pPWFinanceFileInfo)
{
    PWFinanceFileData *somThis =                    /* Get instance data pointer */
          PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile",             /* Set debug info          */
          "pwFinanceFile_QueryInfo");

    strcpy(_szCurrentPassword,                  /* Save user-entered p'word  */
          pPWFinanceFileInfo->szCurrentPassword);
    strcpy(_szPassword,                         /* Save FinanceFile password     */
          pPWFinanceFileInfo->szPassword);
    strcpy(_szUserid,                           /* Save userid             */
          pPWFinanceFileInfo->szUserid);

    return (BOOL) 0;                            /* Return                  */
}

/*
 *
 *   METHOD:   LockFinanceFile                                  PUBLIC
 *
 *   PURPOSE:  Invalidates the current password, thereby locking the FinanceFile.
 *
 *   INVOKED:  From _wpMenuItemSelected
```

```
 *
 */

SOM_Scope BOOL   SOMLINK pwFinanceFile_LockFinanceFile(PWFinanceFile *somSelf)
{
    BOOL bSuccess;

    PWFinanceFileData *somThis =                    /* Get instance data pointer */
          PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile",             /* Set debug info          */
         "pwFinanceFile_QueryInfo");

    strcpy(_szCurrentPassword,"NOPASSWD");       /* Invalid user-entered      */
                                                 /* password                  */
    _wpSetTitle(somSelf,                         /* Set FinanceFile title to      */
               _wpQueryTitle(somSelf) );         /* locked state              */

    bSuccess=_wpSetIcon(somSelf,                 /* Set icon to locked state  */
                        hLockedIcon);

    _wpSaveImmediate(somSelf);                   /* Rember this state         */
    return (BOOL) 0;                             /* Return                    */
}

/*
 *
 *   METHOD:   wpInitData                                PUBLIC
 *
 *   PURPOSE:  Initializes instance data
 *
 *   INVOKED:  By Workplace Shell, upon instantiation of the object instance.
 *
 */

SOM_Scope void   SOMLINK pwFinanceFile_wpInitData(PWFinanceFile *somSelf)
{
    CHAR  ErrorBuffer[100];

    PWFinanceFileData *somThis =                    /* Get instance data pointer */
                PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile",             /* Set debug info          */
                        "pwFinanceFile_wpInitData");

/* set up the debug and tracing */
    SOM_TraceLevel=2;
    SOMOutCharRoutine = myReplacementForSOMOutChar;


    parent_wpInitData(somSelf);                  /* Invoke default processing */

    strcpy(_szCurrentPassword,"password");       /* Initialise FinanceFile in the  */
    strcpy(_szPassword,"password");              /* unlocked state            */
}

/*
 *
 *   METHOD:   wpModifyPopupMenu                         PUBLIC
 *
 *   PURPOSE:  Adds an additional "Lock" item to the object's context menu.
 *             Adds a "Open Finance File" item to the "Open" item
 *   INVOKED:  By Workplace Shell, upon instantiation of the object instance.
 *
 */

SOM_Scope BOOL   SOMLINK pwFinanceFile_wpModifyPopupMenu(PWFinanceFile *somSelf,
               HWND hwndMenu,
               HWND hwndCnr,
               ULONG iPosition)
{
    PWFinanceFileData *somThis =                    /* Get instance data pointer */
                PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile",             /* Set debug info          */
                        "pwFinanceFile_wpModifyPopupMenu");

    _wpInsertPopupMenuItems(somSelf,             /* Insert menu item          */
```

```
                                hwndMenu,           /* Menu handle              */
                                iPosition,          /* Default position         */
                                hmodThisClass,      /* Module handle            */
                                ID_CXTMENU_LOCK,    /* Menu item identifier     */
                                0);                 /* No submenu identifier    */

    _wpInsertPopupMenuItems( somSelf,               /* Insert menu item         */
                             hwndMenu,              /* Menu handle              */
                             0,                     /* at the top!              */
                             hmodThisClass,         /* Module handle            */
                             ID_OPENFinanceFile,    /* Menu item identifier     */
                             WPMENUID_OPEN);        /* Submenu identifier       */


    return(parent_wpModifyPopupMenu(somSelf,        /* Invoke default processing */
                                    hwndMenu,
                                    hwndCnr,
                                    iPosition));
}

/*
 *
 *   METHOD:   wpMenuItemSelected                          PUBLIC
 *
 *   PURPOSE:  Processes the user's selections from the context menu.  The
 *             overridden method processes the added "Lock" & "OPENFinanceFile"
 *             items, and passes all others to the parent method
 *
 *   INVOKED:  By Workplace Shell, upon selection of a menu item by the user.
 *
 */

SOM_Scope BOOL   SOMLINK pwFinanceFile_wpMenuItemSelected(PWFinanceFile *somSelf,
                 HWND hwndFrame,
                 ULONG ulMenuId)
{
    PWFinanceFileData *somThis =                        /* Get instance data pointer */
                 PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile",               /* Set debug info           */
                     "pwFinanceFile_wpMenuItemSelected");

    switch( ulMenuId )                          /* Switch on item identifier */
    {
       case IDM_LOCK:                           /* Lock item selected        */
          _LockFinanceFile(somSelf);                /* Invoke _LockFinanceFile method */
          break;

    /*
     *   We could call wpOpen here, but, if the object is already opened,
     *   the following API determines whether the object should be
     *   resurfaced, or if multiple views are desired.
     *   Must call wpViewObject not wpOpen.  If you use wpOpen, multiple
     *   concurrent views won't work.  User can set object to open multiple views
     *   or switch to.  this function is free if you use wpViewObject.
     */

       case IDM_OPENFinanceFile:                /* Open a view selected      */
          somPrintf("Open my finance file view\n");
          _wpViewObject(somSelf, NULLHANDLE, OPEN_FinanceFile, 0);
          somPrintf("After open my finance file view \n");
          break;


       default:                                 /* All other items           */
          parent_wpMenuItemSelected(somSelf,    /* Invoke default processing */
                                    hwndFrame,
                                    ulMenuId);
          break;
    }
}
/*
 *
 *   METHOD:   wpOpen                                      PUBLIC
 *
 *   PURPOSE:  Only allows a FinanceFile to be opened if the FinanceFile is unlocked, or
```

```
 *            if the user supplies the correct password in response to the
 *            dialog.
 *
 *   INVOKED:  By Workplace Shell, upon selection of the "Open" menu item by
 *             the user.
 *
 */

SOM_Scope HWND   SOMLINK pwFinanceFile_wpOpen(PWFinanceFile *somSelf,
                 HWND hwndCnr,
                 ULONG ulView,
                 ULONG param)
{
    ULONG    ulResult;                          /* Return value              */
    CHAR     szTitle[100];             /* FinanceFile title buffer       */
    PVOID    pCreateParam;
    BOOL     bAllowAccess = FALSE;              /* user is allowed in */

    PWFinanceFileData *somThis =                     /* Get instance data pointer */
                 PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile",             /* Set debug info           */
                        "pwFinanceFile_wpOpen");

    if ((strcmp(_szCurrentPassword,             /* If FinanceFile is locked   */
         _szPassword)) != 0)
    {
       somPrintf("ask for a password\n");
    pCreateParam = malloc( sizeof(ULONG) );     /* Allocate memory to pass a */
                                                /* ULONG to the dialog proc  */
    *((PULONG)pCreateParam) = (ULONG)somSelf;   /* Put the somSelf pointer   */
                                                /* in the CreateParam memory */

    ulResult = WinDlgBox(HWND_DESKTOP,          /* Display password dialog   */
                         HWND_DESKTOP,          /* Desktop is owner          */
                         PasswordDlgProc,       /* Dialog procedure address  */
                         hmodThisClass,         /* Module handle             */
                         ID_DLG_PASSWORD,       /* Dialog resource id        */
                         pCreateParam );        /* Create Param holding the  */
                                                /* pointer to this object    */

    if (ulResult == DID_OK )                    /* If user hit OK button     */
       {
       if ((strcmp(_szCurrentPassword,          /* If password is correct    */
                   _szPassword)) == 0)
          {
          strcpy(szTitle,                       /* Get title string          */
                 _wpQueryTitle(somSelf));
          szTitle[strlen(szTitle)-9] = '\0'; /* Remove <LOCKED>     */
          _wpSetTitle(somSelf,szTitle);         /* Reset title string        */

          _wpSetIcon(somSelf,                   /* Set icon to unlocked      */
                     hUnlockedIcon);            /* state                     */

          /* now we can allow the user access to the object proper ! */
          bAllowAccess = TRUE;

          }
       else                                     /* Password is incorrect     */
          {
          WinMessageBox(HWND_DESKTOP,           /* Display message to user   */
                        HWND_DESKTOP,
                        "Password incorrect. FinanceFile remains locked.",
                        "Password Failed",
                        0,
                        MB_OK |
                        MB_CUAWARNING );
          return((HWND)0);                      /* Return NULL handle        */
          }
       }
    } else {
      bAllowAccess = TRUE;
    }
 somPrintf("now test if allow access\n");
    if (bAllowAccess) {
       switch (ulView) {
```

```
        case OPEN_FinanceFile:
somPrintf("allow access...\n");
          if (!_wpSwitchTo(somSelf, ulView)) {
            /* Create a basic frame and client window for this instance */
            return PWFinanceFileInit(somSelf);
          } /* endif */
          break;

        default:
somPrintf("default processing\n");
          return(parent_wpOpen(somSelf,          /* Allow open to proceed in  */
                               hwndCnr,          /* normal way using default  */
                               ulView,           /* processing                */
                               param));
        } /* endswitch */
    } else {
somPrintf("not allowed access\n");
    } /* endif */
}


/*
 *
 *   METHOD:   wpSetTitle                                    PUBLIC
 *
 *   PURPOSE:  Sets the FinanceFile's title (icon text) to have the phrase <Locked>
 *             as a suffix if the FinanceFile is locked, or removes this suffix if
 *             the FinanceFile is unlocked.
 *
 *   INVOKED:  By wpOpen to set the unlocked state, and by LockFinanceFile to set
 *             the locked state.
 *
 */

SOM_Scope BOOL   SOMLINK pwFinanceFile_wpSetTitle(PWFinanceFile *somSelf,
                 PSZ pszNewTitle)
{
    CHAR szBuf[100];                  /* Character buffer          */

    PWFinanceFileData *somThis =                      /* Get instance data pointer */
                 PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile",              /* Set debug info            */
                        "pwFinanceFile_wpSetTitle");

    strcpy(szBuf,pszNewTitle);                    /* Get current title         */

    if ((strcmp(_szCurrentPassword,               /* If FinanceFile is locked       */
                _szPassword)) != 0)
    {
       if ((strstr(szBuf,"LOCKED")) == NULL)      /* and <LOCKED> not in       */
       {                                          /* current title             */
          strcat(szBuf," <LOCKED>");              /* Add <LOCKED> to title     */
       }
    }
    return (parent_wpSetTitle(somSelf,szBuf));    /* Invoke default processing */
}

/*
 *
 *   METHOD:   wpSetup                                       PUBLIC
 *
 *   PURPOSE:  Sets FinanceFile properties based upon a setup string passed by the
 *             object's creator as part of the WinCreateObject() call.  The
 *             overridden method simply processes the PASSWORD keyword to set
 *             the FinanceFile's password immediately upon instantiation, before
 *             invoking the parent's default processing to handle all other
 *             keywords.
 *
 *   INVOKED:  By the Workplace Shell, upon instantiation of the object
 *             instance.
 *
 */

SOM_Scope BOOL   SOMLINK pwFinanceFile_wpSetup(PWFinanceFile *somSelf,
```

```
                PSZ pszSetupString)
{
    CHAR  pszInitPword[20];            /* Character buffer          */
    BOOL  bFound;                                /* Success flag              */
    ULONG Length;                                /* Returned length           */

    PWFinanceFileData *somThis =                     /* Get instance data pointer */
                PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile",             /* Get debug info            */
                        "pwFinanceFile_wpSetup");

    if (*pszSetupString != '\0')                 /* If string is present      */
       {
       bFound=_wpScanSetupString(somSelf,        /* Parse setup string to     */
                                 pszSetupString, /* find PASSWORD keyword     */
                                 "PASSWORD",
                                 pszInitPword,   /* Buffer for keyword value  */
                                 &Length);       /* Length of returned string */

       if (bFound)
          {
          strcpy(_szPassword,                    /* Initialize FinanceFile        */
                 pszInitPword);                  /* password                  */
          strcpy(_szCurrentPassword,             /* Initialize user-entered   */
                 pszInitPword);                  /* password                  */
          }
       }
    return(parent_wpSetup(somSelf,               /* Invoke default processing */
                          pszSetupString));
}

/*
 *
 *   METHOD:   wpSaveState                                PUBLIC
 *
 *   PURPOSE:  Saves the object instance's persistent state data.  The
 *             overridden method simply saves the password data, then invokes
 *             the parent's default processing to handle any other instance
 *             data defined by ancestor classes.
 *
 *   INVOKED:  By the Workplace Shell, when the object becomes dormant.
 *
 */

SOM_Scope BOOL   SOMLINK pwFinanceFile_wpSaveState(PWFinanceFile *somSelf)
{
    PWFinanceFileData *somThis =                     /* Get instance data pointer */
                PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile",             /* Set debug info            */
                        "pwFinanceFile_wpSaveState");

    _wpSaveString(somSelf,                       /* Save FinanceFile password     */
                  "PWFinanceFile",                   /* Class name                */
                  1L,                            /* Class-defined key         */
                  _szPassword);                  /* String to be saved        */
    _wpSaveString(somSelf,                       /* Save user-entered p'word  */
                  "PWFinanceFile",                   /* Class name                */
                  2L,                            /* Class-defined key         */
                  _szCurrentPassword);           /* String to be saved        */

    return(parent_wpSaveState(somSelf));         /* Invoke default processing */
}

/*
 *
 *   METHOD:   wpRestoreState                             PUBLIC
 *
 *   PURPOSE:  Restores the object instance's persistent state data.  The
 *             overridden method simply restores the password data, then
 *             invokes the parent's default processing to handle any other
 *             instance data defined by ancestor classes.
 *
 *   INVOKED:  By the Workplace Shell, when the object becomes awake.
 *
 */
```

```
SOM_Scope BOOL   SOMLINK pwFinanceFile_wpRestoreState(PWFinanceFile *somSelf,
                 ULONG ulReserved)
{
    ULONG ulRetLength;                          /* Length of returned string */

    PWFinanceFileData *somThis =                     /* Get instance data pointer */
                 PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile",            /* Set debug info          */
                        "pwFinanceFile_wpRestoreState");

    _wpRestoreString(somSelf,                   /* Restore FinanceFile password   */
                     "PWFinanceFile",                /* Class name                 */
                     1L,                        /* Class-defined key         */
                     _szPassword,               /* String to be restored     */
                     &ulRetLength);             /* Length of returned string */
    _wpRestoreString(somSelf,                   /* Restore user-entered pwd  */
                     "PWFinanceFile",                /* Class name                 */
                     2L,                        /* Class-defined key         */
                     _szCurrentPassword,        /* String to be restored     */
                     &ulRetLength);             /* Length of returned string */

    if ((strcmp(_szCurrentPassword,             /* If FinanceFile is locked       */
                _szPassword)) != 0)
    {
       _wpSetIcon(somSelf, hLockedIcon);        /* Set icon to locked state  */
    }

    return(parent_wpRestoreState(somSelf,       /* Invoke default processing */
                                 ulReserved));
}

/*
 *
 *   METHOD:   wpSetIcon                                PUBLIC
 *
 *   PURPOSE:  This class method returns the handle to the correct icon for
 *             the object.
 *
 *   INVOKED:
 *
 */

SOM_Scope BOOL   SOMLINK pwFinanceFile_wpSetIcon(PWFinanceFile *somSelf,
                 HPOINTER hptrNewIcon)
{
    PWFinanceFileData *somThis = PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile","pwFinanceFile_wpSetIcon");

    if ((strcmp(_szCurrentPassword,         /* If password is correct     */
                _szPassword)) == 0)
    {
       return (parent_wpSetIcon(somSelf,
                     hUnlockedIcon));       /* return pointer to unlocked */
    }
    else                                    /* otherwise                  */
    {
       return (parent_wpSetIcon(somSelf,
                     hLockedIcon));         /* return locked icon pointer */
    }
}

#undef SOM_CurrentClass
#define SOM_CurrentClass SOMMeta
/*
 *
 *   METHOD:   wpclsQueryTitle                          PUBLIC
 *
 *   PURPOSE:  This class method returns the default FinanceFile title for any
 *             instance of the password protected FinanceFile class.  This title
 *             is used if a title is not supplied in the WinCreateObject()
 *             call.
 *
 *   INVOKED:  By the Workplace Shell, upon instantiation of the object
 *             instance.
```

```
 *
 */

SOM_Scope PSZ   SOMLINK pwFinanceFilecls_wpclsQueryTitle(M_PWFinanceFile *somSelf)
{
    /* M_PWFinanceFileData *somThis = M_PWFinanceFileGetData(somSelf); */

    M_PWFinanceFileMethodDebug("M_PWFinanceFile",          /* Set debug info           */
                        "pwFinanceFilecls_wpclsQueryTitle");

    return(DefaultClassTitle);                    /* Return default title      */
}

/*
 *
 *   METHOD:   wpclsQueryIcon                              PUBLIC
 *
 *   PURPOSE:  This class method returns the handle to the default icon for
 *             the class.  This method is not used in the current version,
 *             but could be used if different icons are to be used for the
 *             locked and unlocked states.
 *
 *   INVOKED:
 *
 */

SOM_Scope HPOINTER   SOMLINK pwFinanceFilecls_wpclsQueryIcon(M_PWFinanceFile *somSelf)
{
    /* M_PWFinanceFileData *somThis = M_PWFinanceFileGetData(somSelf); */
    PWFinanceFileData *somThis =                       /* Get instance data pointer */
                PWFinanceFileGetData(somSelf);

    M_PWFinanceFileMethodDebug("M_PWFinanceFile",          /* Set debug info           */
                        "pwFinanceFilecls_wpclsQueryIcon");

    return(hUnlockedIcon);
}

/*
 *
 *   METHOD:   wpclsInitData                               PUBLIC
 *
 *   PURPOSE:  This class method allows the initialization of any class data
 *             items.  The overridden method simply obtains a module handle
 *             to be used when accessing Presentation Manager resources, then
 *             invokes the parent's default processing.
 *
 *   INVOKED:  By the Workplace Shell, upon loading the class DLL.
 *
 */

SOM_Scope void   SOMLINK pwFinanceFilecls_wpclsInitData(M_PWFinanceFile *somSelf)
{
    APIRET apiret;

/*    M_PWFinanceFileData *somThis = M_PWFinanceFileGetData(somSelf); */

    M_PWFinanceFileMethodDebug("M_PWFinanceFile",          /* Set debug info           */
                        "pwFinanceFilecls_wpclsInitData");

    apiret = DosQueryModuleHandle("PWFin", & hmodThisClass);

    if (apiret == NO_ERROR)
       somPrintf("Handle OK\n");
    else
       {
       somPrintf("Handle Not OK tell me Why???\n");
       }

    hLockedIcon=WinLoadPointer(HWND_DESKTOP,     /* Load icons               */
                               hmodThisClass,
                               ID_LOCK);
    hUnlockedIcon=WinLoadPointer(HWND_DESKTOP,
                                 hmodThisClass,
```

```
                                ID_UNLOCK);

    parent_wpclsInitData(somSelf);              /* Invoke default processing */
}

/*
 *
 *   METHOD:   wpclsUnInitData                                PUBLIC
 *
 *   PURPOSE:  This class method allows the release of any class data items
 *             or resources.  The overridden method releases the module handle
 *             obtained by wpclsInitData, then invokes the parent's default
 *             processing.
 *
 *   INVOKED:  By the Workplace Shell, upon unloading the class DLL.
 *
 */

SOM_Scope void   SOMLINK pwFinanceFilecls_wpclsUnInitData(M_PWFinanceFile *somSelf)
{
    /* M_PWFinanceFileData *somThis = M_PWFinanceFileGetData(somSelf); */

    M_PWFinanceFileMethodDebug("M_PWFinanceFile",          /* Set debug info           */
                        "pwFinanceFilecls_wpclsUnInitData");

    WinDestroyPointer(hLockedIcon);             /* Free icons               */
    WinDestroyPointer(hUnlockedIcon);

    parent_wpclsUnInitData(somSelf);
}

/*****************************************************************************/
/*                                                                           */
/* PROCEDURE NAME:  PasswordDlgProc                                          */
/*                                                                           */
/* description:  Dialog procedure for password dialog                        */
/*                                                                           */
/* invoked:  By Presentation Manager, in response to FinanceFile issuing          */
/*           WinDlgBox() call from _wpOpen method.                           */
/*                                                                           */
/*****************************************************************************/
MRESULT EXPENTRY PasswordDlgProc(HWND hwndDlg,
                                 ULONG msg,
                                 MPARAM mp1,
                                 MPARAM mp2)
{
   PWFinanceFile *aPWF;                             /* Define SOM pointer       */
   PWF_INFO pwFinanceFileInfo;                      /* Define password structure */
   CHAR     szTemp[100];              /* Character buffer         */

   switch (msg)                                 /* Determine message class   */
   {
      case WM_INITDLG:                          /* Dialog being initialized  */

         WinSetWindowULong(hwndDlg,             /* Store SOM pointer in      */
                           QWL_USER,            /* window word QWL_USER      */
                           *((PULONG)mp2));
         free(mp2);                             /* Free Create Param memory  */
         break;

      case WM_COMMAND:                          /* User hit a pushbutton     */

         aPWF = (PWFinanceFile *)WinQueryWindowULong(hwndDlg, QWL_USER);

         _QueryInfo(aPWF,                       /* Get password data         */
                    &pwFinanceFileInfo);

         switch (SHORT1FROMMP(mp1))             /* Which button was hit?     */
         {
            case DID_OK:                        /* Okay button               */

               WinQueryDlgItemText(hwndDlg,     /* Get text from entryfield  */
                           ID_EF_PASSWORD,
                           sizeof(szTemp),
                           (PSZ)szTemp);
```

```
                                                /* Copy to password data     */
                strcpy(pwFinanceFileInfo.szCurrentPassword, szTemp);

                _SetInfo(aPWF,&pwFinanceFileInfo);     /* Set instance data         */
                WinDismissDlg(hwndDlg,DID_OK);    /* Dismiss dialog            */
                break;

            case DID_CANCEL:                      /* Cancel button hit         */

                WinDismissDlg(hwndDlg,            /* Dismiss dialog            */
                              DID_CANCEL);
                break;
            }
            return(MRESULT)TRUE;                  /* Return from WM_COMMAND    */
            break;
        }
        return(WinDefDlgProc(hwndDlg,             /* Invoke default PM message */
                             msg,                 /* handling                  */
                             mp1,
                             mp2));
}

/*
 *
 *   METHOD:   wpDraggedOverObject                          PUBLIC
 *
 *   PURPOSE:  Checks to see that the file is unlocked if user wants to
 *             drop the object on a program.
 *
 *   INVOKED:  By Workplace Shell, upon instantiation of the object instance.
 *
 */

//SOM_Scope MRESULT   SOMLINK pwFinanceFile_wpDraggedOverObject(PWFinanceFile *somSelf,
 MRESULT   SOMLINK pwFinanceFile_wpDraggedOverObject(PWFinanceFile *somSelf,
                WPObject *DraggedOverObject)
{
    CLASS          PWFinanceFileClass;

    PWFinanceFileData *somThis = PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile","pwFinanceFile_wpDraggedOverObject");

    somPrintf("pwFinanceFile_wpDraggedOverObject");/* DHP */


    PWFinanceFileClass = _somClassFromId( SOMClassMgrObject,
                         SOM_IdFromString("PWFinanceFile") );

//    somPrintf(PWFinanceFileClass);

    if ((strcmp(_szCurrentPassword,               /* If FinanceFile is locked      */
                _szPassword)) != 0)
    {
        return (MRFROM2SHORT(DOR_NODROP,DO_UNKNOWN));
    }
    if (_somIsA(DraggedOverObject,_WPProgram)     ||
        _somIsA(DraggedOverObject, PWFinanceFileClass) ||
        _somIsA(DraggedOverObject,_WPProgramFile)) {

        return (MRFROM2SHORT(DOR_DROP,DO_MOVE));
      } /* endif */

    return (parent_wpDraggedOverObject(somSelf,DraggedOverObject));
}


/*
 *
 *   METHOD:   wpAddFileTypePage                            PUBLIC
 *
 *   PURPOSE:  This class method replaces the type page with the it's own one
 *             that only allows the FinanceFile Types.
 *
```

```
 *   INVOKED:  By the Workplace Shell, upon unloading the class DLL.
 *
 */

SOM_Scope ULONG   SOMLINK pwFinanceFile_wpAddFileTypePage(PWFinanceFile *somSelf,
                HWND hwndNotebook)
{
   PSZ psztype;
    PWFinanceFileData *somThis = PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile","pwFinanceFile_wpAddFileTypePage");


// PAGEINFO pageinfo;
//
//    memset((PCH)&pageinfo,0,sizeof(PAGEINFO));
//    pageinfo.cb                 = sizeof(PAGEINFO);
//    pageinfo.hwndPage           = NULLHANDLE;
//    pageinfo.usPageStyleFlags   = BKA_MAJOR;
//    pageinfo.usPageInsertFlags  = BKA_FIRST;
//    pageinfo.pfnwp              = DashBoardDlgProc;
//    pageinfo.resid              = hmod;
//    pageinfo.dlgid              = IDD_PWFILETYPES;
//    pageinfo.pszName            = "File Type";
//    pageinfo.pCreateParams      = somSelf;
//    pageinfo.idDefaultHelpPanel = ID_HELP_DASHBOARD;
//    pageinfo.pszHelpLibraryName = szHelpLibrary;
//
//    return _wpInsertSettingsPage( somSelf, hwndNotebook, &pageinfo );

//  psztype = _wpQueryType(somSelf);
//  somPrintf("Data Type = >");
//  somPrintf(psztype);
//  somPrintf("<\n");
    return (SETTINGS_PAGE_REMOVED);
//    return (parent_wpAddFileTypePage(somSelf,hwndNotebook)); // do nothing

}

/*
 *
 *   METHOD:   wpDragOver                                    PUBLIC
 *
 *   PURPOSE:  Checks to see that the object being dragged over me is also
 *             either of my class or derived from me.
 *
 *   INVOKED:  By Workplace Shell, upon instantiation of the object instance.
 *
 */

SOMAny *queryObjectFromDragItem( PDRAGITEM pDragItem)
{
   WPObject *Object=NULL;

   if ( DrgVerifyRMF (pDragItem, "DRM_OBJECT", NULL)) {
     Object = OBJECT_FROM_DRAGITEM(pDragItem);
     somPrintf("OBJECT!!\n");
   } else {
        somPrintf("NOT AN OBJECT!!\n");
/* add code in here to deal with non-objects, eg datafiles, and lines */
/* selected from a listbox from a PWFinanceFile etc etc */
   } /* endif */
   return(Object);
}

SOM_Scope MRESULT   SOMLINK pwFinanceFile_wpDragOver(PWFinanceFile *somSelf,
                HWND hwndCnr,
                PDRAGINFO pdrgInfo)
{
    ULONG    ulItemCount         =0;
    ULONG    ulItem              =0;
    CLASS    PWFinanceFileClass;
    SOMAny  *ObjectBeingDragged;
    MRESULT  mr;
    USHORT   dropOperation,
             dropIndicator;
```

```
    PWFinanceFileData *somThis = PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile","pwFinanceFile_wpDragOver");

    if ((strcmp(_szCurrentPassword,             /* If FinanceFile is locked      */
                _szPassword)) != 0)
    {
        return (MRFROM2SHORT(DOR_NODROP,DO_UNKNOWN));
    }

/* for each of the items being dragged, check to see that they are all */
/* derived from PWFinanceFileClass                                     */

    PWFinanceFileClass = _somClassFromId( SOMClassMgrObject,
                         SOM_IdFromString("PWFinanceFile") );

    /* firstly will all the source object(s) pass my parents tests? */
    mr = parent_wpDragOver(somSelf,hwndCnr,pdrgInfo);
    dropIndicator = SHORT1FROMMR(mr);
    dropOperation = SHORT2FROMMR(mr);

    if (dropIndicator != DOR_NEVERDROP)
    { somPrintf("passed parents testing\n");
      /* passed the parent's tests, so unless it fails this object's */
      /* tests we will allow the DROP                                */
      dropIndicator = DOR_DROP;
      dropOperation = DO_COPY;

      /* how many items are being dragged ? */
      ulItemCount = DrgQueryDragitemCount(pdrgInfo);

      /* search through the objects and abort if we find any that aren't derived */
      /* from PWFinanceFileClass                                                  */
  somPrintf("Number of Items being dragged = %i.\n",ulItemCount);
      for (ulItem=0; ulItem<ulItemCount; ulItem++) {
        PDRAGITEM pDragItem; /* temporary variable*/

        /* get one of the one or more drag items that we are receiving */
        pDragItem = DrgQueryDragitemPtr(pdrgInfo, ulItem);

        ObjectBeingDragged = queryObjectFromDragItem(pDragItem);

        if (ObjectBeingDragged) {
          if (!_somIsA(ObjectBeingDragged,PWFinanceFileClass)) {
somPrintf("Object %i, is rejected because it is not derived from PWFinanceFileClass\n",ulItem);
            return (MRFROM2SHORT(DOR_NEVERDROP,DO_UNKNOWN));
          } /* endif */
somPrintf("Object %i, is acceptable for dropping\n",ulItem);
        } else {
somPrintf("Object %i, is not a WPS object\n",ulItem);
//          return (MRFROM2SHORT(DOR_NEVERDROP,DO_UNKNOWN)); /* not an object */
        } /* endif */

      } /* for */
    } /* endif */
   return (MRFROM2SHORT(dropIndicator, dropOperation));

/* if we do the following line, the icon stays at the do not do symbol, why */
/* return (parent_wpDragOver(somSelf,hwndCnr,pdrgInfo)); */
}

/*
 *
 *   METHOD:   wpPrintObject                              PUBLIC
 *
 *   PURPOSE:  This class method allows this object to format it's output for
 *             printing.
 *
 *   INVOKED:
 *
 */

SOM_Scope BOOL   SOMLINK pwFinanceFile_wpPrintObject(PWFinanceFile *somSelf,
                 PPRINTDEST pPrintDest,
```

```
                ULONG ulReserved)
{
   BOOL worked;

    PWFinanceFileData *somThis = PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile","pwFinanceFile_wpPrintObject");
somPrintf("Yes this is the new PrintObject, printing is disabled\n");
      /* disable printing for now */
      return(FALSE);
//      return(parent_wpPrintObject(somSelf,pPrintDest,ulReserved));
}

/*
 *
 *   METHOD:   wpDrop                                        PUBLIC
 *
 *   PURPOSE:  To receive a dropped object.
 *
 *   INVOKED:  By Workplace Shell, when another object has been dropped on
 *             this object.
 *
 */

SOM_Scope MRESULT   SOMLINK pwFinanceFile_wpDrop(PWFinanceFile *somSelf,
                HWND hwndCnr,
                PDRAGINFO pdrgInfo,
                PDRAGITEM pdrgItem)
{

    CHAR          szName[CCHMAXPATH];
    CHAR          szPath[CCHMAXPATH];
    ULONG         cbPath = CCHMAXPATH;
    ULONG         ulItemCount         =0;
    ULONG         ulItem              =0;
    CLASS         PWFinanceFileClass;
    SOMAny       *ObjectBeingDragged;
    MRESULT       mr;
    USHORT        dropOperation,
                  dropIndicator;
    BOOL          flPrepared = TRUE; // Assume we do not need to do a prepare
    BOOL          flRendering = FALSE;
    PDRAGTRANSFER pDragTransfer;

    PWFinanceFileData *somThis = PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile","pwFinanceFile_wpDrop");

    if ((strcmp(_szCurrentPassword,       /* If FinanceFile is NOT locked */
                _szPassword)) == 0)
    {

  /* make sure we are not dragging ourselves, and dropping onto ourselves */
     if (pdrgInfo->hwndSource != hwndCnr)
     {

     /* for each of the items being dropped, check to see that they are all */
     /* derived from PWFinanceFileClass                                     */

     PWFinanceFileClass = _somClassFromId( SOMClassMgrObject,
                          SOM_IdFromString("PWFinanceFile") );

      /* passed the parent's tests, so unless it fails this object's */
      /* tests we will allow the DROP                                */
      dropIndicator = DOR_DROP;
      dropOperation = DO_COPY;

      /* how many items are being dragged ? */
      ulItemCount = DrgQueryDragitemCount(pdrgInfo);

      /* search through the objects and abort if we find any that aren't derived */
      /* from PWFinanceFileClass                                                  */
      somPrintf("Number of Items being dropped = %i.\n",ulItemCount);
      for (ulItem=0; ulItem<ulItemCount; ulItem++) {
        PDRAGITEM pDragItem; /* temporary variable*/

        /* get one of the one or more drag items that we are receiving */
```

```
  pDragItem = DrgQueryDragitemPtr(pdrgInfo, ulItem);

  ObjectBeingDragged = queryObjectFromDragItem(pDragItem);

  if (ObjectBeingDragged) {
    if (!_somIsA(ObjectBeingDragged,PWFinanceFileClass)) {
      somPrintf("Object %i, is rejected for drop because it ",ulItem);
      somPrintf("is not derived from PWFinanceFileClass\n");
    } else {
      somPrintf("Object %i, is acceptable for dropping, by wpDROP\n",ulItem);
    } /* endif */
  } else {
    somPrintf("Object %i, is not a WPS object, can we render it\n",ulItem);

/* start of code to render item */

if( DrgVerifyRMF (pDragItem, "DRM_OS2FILE", NULL) )
{
   somPrintf("An OS2FILE rendering method!\n");
   /* Protocol allows the source object to propose a target name...
    *
    * If it does, then try to use it, if it does not, then
    * try to use the source name, if present.  Finally, just
    * make up our own name...
    */
   if (pDragItem->hstrTargetName &&
       DrgQueryStrNameLen(pDragItem->hstrTargetName) )
   {
      DrgQueryStrName(pDragItem->hstrTargetName,
                      sizeof(szName),szName);
      somPrintf("Source proposes the target filename\n");
   }
   else
   {
      if (pDragItem->hstrSourceName &&
         DrgQueryStrNameLen(pDragItem->hstrSourceName))
      {
         DrgQueryStrName(pDragItem->hstrSourceName,
                         sizeof(szName),szName);
      somPrintf("Source proposes the source filename\n");

      }
      else
      {
         szName[0] = '\0';
         somPrintf("no source, nor target name\n");
      }
   }

   /* Allocate and initialize a drag transfer structure
    */
   somPrintf("allocating pDragtransfer structure\n");
   pDragTransfer = DrgAllocDragtransfer(1);

   if (pDragTransfer)
   {
      somPrintf("pDragtransfer structure allocated ok\n");
      /* create a file or directory now to get its true name
       */
      ObjectBeingDragged = _wpclsNew( _WPDataFile,
                           szName,
                           NULL,
                           _wpclsQueryFolder(_WPDataFile,"<WP_NOWHERE>",TRUE),
                           TRUE );

      if (ObjectBeingDragged)
      {
         somPrintf("ObjectBeingDragged has been successfully allocated\n");
         _wpQueryRealName(ObjectBeingDragged,szPath,&cbPath,TRUE);
         somPrintf("The ObjectBeingDragged filename is %s\n",szPath);

         /* fill in the struct now
          */
         pDragTransfer->cb              = sizeof(DRAGTRANSFER);
         pDragTransfer->hwndClient      = hwndCnr;
```

```
                pDragTransfer->pditem           = pDragItem;
                pDragTransfer->hstrSelectedRMF  =
                                DrgAddStrHandle("<DRM_OS2FILE,DRF_UNKNOWN>");
                pDragTransfer->hstrRenderToName = DrgAddStrHandle( szPath );
                pDragTransfer->ulTargetInfo     = 0L;
                pDragTransfer->usOperation      = pdrgInfo->usOperation;
                pDragTransfer->fsReply          = 0;


                /* Now, if the source wants prepared, do it...
                 */
                if (pDragItem->fsControl & DC_PREPARE)
                {
                   somPrintf("Source wants prepared\n");
                   flPrepared = (BOOL)DrgSendTransferMsg(pdrgInfo->hwndSource,
                                                         DM_RENDERPREPARE,
                                                         (MPARAM)pDragTransfer,
                                                         (MPARAM)NULL);
                } else {
                   somPrintf("Source does not want prepared\n");
                }
                /* See if either we did not need to send a RENDERPREPARE, or
                 * we have successfully done so...
                 */
                if (flPrepared)
                {
                   somPrintf("not prepared\n");
                   /* Tell the source object where to put the file.
                    */
                   flRendering = (BOOL)DrgSendTransferMsg(pDragItem->hwndItem,
                                                          DM_RENDER,
                                                          (MPARAM)pDragTransfer,
                                                          (MPARAM)NULL);
                   if (!flRendering)
                   {
                      /* The partner object did not render, so delete
                       * the object we just created.
                       */
                      _wpFree(ObjectBeingDragged);
                      somPrintf("not rendering, we are deleting the object we just created\n");
                   } else {
                      somPrintf("rendering\n");
                   }
                }
                else
                {
                   somPrintf("Our partner wanted us to send him a prepare, and");
                   somPrintf("now has changed his mind about things..., ABORT\n");

                   /* Our partner wanted us to send him a prepare, and
                    * now has changed his mind about things...
                    * We cannot even send him an end conversation, as
                    * we do not know that the hwnd is any good.
                    *
                    * For now, we will treat this as an error.
                    */
                   mr = (MRESULT)RC_DROP_ERROR;
                }
             } else {
               somPrintf("ObjectBeingDragged has NOT been successfully allocated\n");
             }
             if (flRendering)
             {
                mr = RC_DROP_RENDERING;
             }
             else
             {
                DrgDeleteStrHandle( pDragTransfer->hstrRenderToName );
                DrgFreeDragtransfer( pDragTransfer );
             }
          }
       } else {
          somPrintf("Not an OS2FILE rendering methid\n");
       }
```

```
        } /* endif */

      } /* for */
    } else {
       somPrintf("we are trying to drop onto ourselves, not allowed\n");
    } /* endif */
  } else {
     somPrintf("LOCKED, drop is disallowed\n");
  } /* endif */

  return((HRESULT) NULL);

}

/*
 *
 *   METHOD:   wpFilterPopupMenu                          PUBLIC
 *
 *   PURPOSE:  This class method is called when the user asks for the context
 *             (popup) menu.
 *
 *   INVOKED:
 *
 */

SOM_Scope ULONG   SOMLINK pwFinanceFile_wpFilterPopupMenu(PWFinanceFile *somSelf,
                ULONG ulFlags,
                HWND hwndCnr,
                BOOL fMultiSelect)
{ ULONG ulPopupFlags;

    PWFinanceFileData *somThis = PWFinanceFileGetData(somSelf);
    PWFinanceFileMethodDebug("PWFinanceFile","pwFinanceFile_wpFilterPopupMenu");
    somPrintf("We were passed %o\n",ulFlags);

    /* first find out what our ancestors have done! */
//    ulPopupFlags = parent_wpFilterPopupMenu(somSelf,ulFlags,hwndCnr,fMultiSelect);
//    somPrintf("Our ancestor changed this to %o\n",ulPopupFlags);
      ulPopupFlags = ulFlags;

    /* now what have the done to the "Create another" menu item */
    if ((ulPopupFlags & CTXT_NEW) == CTXT_NEW) {
       /* the "Create another" menu item is on our Popup, so remove it */
       somPrintf("'Create another' menu item is being removed\n");
       ulPopupFlags = ulPopupFlags & ~CTXT_NEW;
    } else {
       /* the "Create another" menu item is NOT on our Popup, so add it */
       somPrintf("'Create another' menu item is being added\n");
       ulPopupFlags = ulPopupFlags | CTXT_NEW;
    } /* endif */
    somPrintf("We changed this to %o\n",ulPopupFlags);

    return(ulPopupFlags);
}

/*************************  ORDINARY CODE SECTION  ***************************
*****                                                                   *****
*****                 Any non-method code should go here.               *****
*****                                                                   *****
****************************************************************************/
#undef SOM_CurrentClass


/****************************************************************************
*                                                                           *
*       ROUTINE:    PWFinanceFileInit()                                     *
*                                                                           *
*       DESCRIPTION:  PWFinanceFile Inisialisation                          *
*                                                                           *
*       RETURNS:    Handle of PWFinanceFile frame window, NULL if error     *
*                                                                           *
****************************************************************************/
HWND PWFinanceFileInit (PWFinanceFile* somSelf)
{
```

```
HAB  hab;                                        /* PM anchor block handle */
HWND hwndFrame = NULLHANDLE;                        /* Frame window handle */
HWND hwndClient = NULLHANDLE;
PWINDOWDATA pWindowData;
BOOL fSuccess;
SWCNTRL    swcEntry;                                      /* Switch Entry */
FRAMECDATA flFrameCtlData;                               /* Frame Ctl Data */

somPrintf("PWFinanceFileInit\n");

hab = WinQueryAnchorBlock(HWND_DESKTOP);
if (!WinRegisterClass( hab , szFinanceFileWindowClass, (PFNWP)FinanceFileWndProc ,
                       CS_SIZEREDRAW | CS_SYNCPAINT, sizeof(pWindowData)))
{
   somPrintf("FinanceFileInit Failure in WinRegisterClass\n");
   return NULLHANDLE ;
}

/*
 *   Allocate some instance specific data in Window words of Frame window.
 *   This will ensure our window procedure can use this object's methods
 *   (our window proc isn't passed a * somSelf pointer).
 */
pWindowData = (PWINDOWDATA) _wpAllocMem(somSelf, sizeof(*pWindowData), NULL);

if (!pWindowData)
{
   somPrintf("FinanceFileInit wpAllocMem failed to allocate pWindowData\n");
   return NULLHANDLE;
}

memset((PVOID) pWindowData, 0, sizeof(*pWindowData));
pWindowData->cb = sizeof(*pWindowData);             /* first field = size */
pWindowData->somSelf = somSelf;

/* Create a frame window
 */
flFrameCtlData.cb            = sizeof( flFrameCtlData );
flFrameCtlData.flCreateFlags = FCF_SIZEBORDER | FCF_TITLEBAR | FCF_SYSMENU |
                               FCF_MINMAX ;
flFrameCtlData.hmodResources = hmodThisClass;
flFrameCtlData.idResources   = ID_UNLOCK;

hwndFrame =                                         /* create frame window */
WinCreateWindow(
   HWND_DESKTOP,              /* parent-window handle                    */
   WC_FRAME,                  /* pointer to registered class name        */
   _wpQueryTitle(somSelf),    /* pointer to window text                  */
   0,                         /* window style                            */
   0, 0, 0, 0,                /* position of window                      */
   NULLHANDLE,                /* owner-window handle                     */
   HWND_TOP,                  /* handle to sibling window                */
   (USHORT) ID_FRAME,         /* window identifier                       */
   (PVOID) &flFrameCtlData,   /* pointer to buffer                       */
   NULL);      ;              /* pointer to structure with pres. params. */

if (!hwndFrame)
{
   somPrintf("FinanceFileInit Failure in WinCreateWindow\n");
   return NULLHANDLE;
}
hwndClient =        /* use WinCreateWindow so we can pass pres params */
WinCreateWindow(
   hwndFrame,                 /* parent-window handle                    */
   szFinanceFileWindowClass,        /* pointer to registered class name        */
   NULL,                      /* pointer to window text                  */
   0,                         /* window style                            */
   0, 0, 0, 0,                /* position of window                      */
   hwndFrame,                 /* owner-window handle                     */
   HWND_TOP,                  /* handle to sibling window                */
   (USHORT)FID_CLIENT,        /* window identifier                       */
   pWindowData,               /* pointer to buffer                       */
   NULL);                     /* pointer to structure with pres. params. */
```

```
   if (!hwndClient)
   {
      WinDestroyWindow(hwndFrame);
      return NULLHANDLE;
   }

   WinSendMsg(hwndFrame,WM_SETICON,MPFROMP(_wpQueryIcon(somSelf)),NULL);
   WinSetWindowText(WinWindowFromID(hwndFrame,(USHORT)FID_TITLEBAR),
                                                  _wpQueryTitle(somSelf));

   /*
    * Restore the Window Position
    */
   fSuccess =
   WinRestoreWindowPos(
      szFinanceFileClassTitle,                                  /* class title */
      _wpQueryTitle(somSelf),                            /* object title */
      hwndFrame);

   if (!fSuccess)
   {
    SWP        swp;

      /* Get the dimensions and the shell's suggested
       * location for the window
       */
      WinQueryTaskSizePos(hab,0,&swp);

      /* Set the frame window position
       */
      swp.fl              = SWP_SIZE|SWP_MOVE|SWP_RESTORE|SWP_ZORDER;
      WinSetWindowPos(hwndFrame, HWND_TOP, swp.x, swp.y, swp.cx,
                      swp.cy, swp.fl);
   }

   WinShowWindow(hwndFrame,TRUE);

   return hwndFrame;                                        /* success */

}  /* end FinanceFileInit() */


/*****************************************************************************
*
*   FinanceFileWndProc()
*
*   DESCRIPTION: FinanceFile Window Procedure
*
*****************************************************************************/
MRESULT EXPENTRY FinanceFileWndProc( HWND hwnd, ULONG msg, MPARAM mp1, MPARAM mp2 )
{
  ULONG         MenuId;
  PWINDOWDATA   pWindowData;
  HWND          hwndFrame;
  CHAR          acBuffer[10];
  BOOL          fSuccess;
  CHAR          szPath[CCHMAXPATH];
  ULONG         cbPath = CCHMAXPATH;


   hwndFrame = WinQueryWindow(hwnd, QW_PARENT);

   switch( msg )
   {
      case WM_CREATE:

         pWindowData = (PWINDOWDATA) mp1;

         if (pWindowData == NULL)
         {
            somPrintf("FinanceFileWndProc:WM_CREATE couldn't get window words");
            return FALSE;
         }
         /*
          *   Fill in the class view/usage details and window specific data
```

```
     *   for this instance.
     *   Must create useitem, add it to the object's use list and register
     *   the view
     */

    pWindowData->UseItem.type     = USAGE_OPENVIEW;
    pWindowData->ViewItem.view    = OPEN_FinanceFile;
    /*
     * Must be frame.  be careful because this procedure is for the client.
     * must get parent and pass that as ViewItem.handle.
     */
    pWindowData->ViewItem.handle = hwndFrame;
    pWindowData->x               = 10;
    pWindowData->y               = 10;
    pWindowData->xDir            = 0;
    pWindowData->yDir            = 0;

    /*
     *   Set window pointer with object pointer and instance view info.
     *   Then add view to the in-use list so wpSwitchTo works.
     */
    WinSetWindowPtr(hwnd, QWL_USER, pWindowData);
    /*
     * _wpAddToObjUseList will tell the shell to store the view in
     * the internal linked list for the object to enable wpSwitchTo and other
     * methods to find the view.  The shell will also subclass the view window
     * this gives you title bar context menu when you call wpRegisterView.
     * wpRegisterView also puts the view in the window list and sets up
     * the title bar like:  "Object Title - View Title"
     */

    _wpAddToObjUseList(pWindowData->somSelf,&pWindowData->UseItem);
    _wpRegisterView(pWindowData->somSelf, hwndFrame,
                                        _wpQueryTitle(pWindowData->somSelf));
    WinSetFocus( HWND_DESKTOP, hwndFrame);

    /* what is the filename of the file */
    if (_wpQueryRealName(pWindowData->somSelf,szPath,&cbPath,TRUE))
       {
          somPrintf("File name is %s, size %i \n",szPath, cbPath);
       } else {
          somPrintf("Failed to get filename\n");
       } /* endif */

    break;

  case WM_COMMAND:

    break;

  case WM_PAINT:
     pWindowData = (PWINDOWDATA) WinQueryWindowPtr(hwnd, QWL_USER);

     if (pWindowData == NULL)
     {
        somPrintf("FinanceFileWndProc:WM_PAINT couldn't get window words\n");
        return FALSE;
     }
     else
     {
      HPS    hps;
      RECTL  rectl;

       hps = WinBeginPaint( hwnd, (HPS)NULLHANDLE, &rectl);
       WinFillRect( hps, &rectl, SYSCLR_WINDOW);
       WinEndPaint( hps );
     }
     break;

  case WM_CLOSE:
     {
      HAB hab;

        hab = WinQueryAnchorBlock(HWND_DESKTOP);
```

```
                pWindowData = (PWINDOWDATA) WinQueryWindowPtr(hwnd, QWL_USER);

                if (pWindowData == NULL)
                {
                   somPrintf("FinanceFileWndProc:WM_CLOSE couldn't get window words\n");
                   return FALSE;
                }
                fSuccess =
                WinStoreWindowPos(szFinanceFileClassTitle,_wpQueryTitle(pWindowData->somSelf),
                                                                          hwndFrame);
                /*
                 * Must remove from obj use list when window is closed.  (can be done
                 * on WM_DESTROY instead)
                 */
                _wpDeleteFromObjUseList(pWindowData->somSelf,&pWindowData->UseItem);
                _wpFreeMem(pWindowData->somSelf,(PBYTE)pWindowData);

                WinDestroyWindow ( hwndFrame ) ;
             }
             break;

          default:
             return WinDefWindowProc( hwnd, msg, mp1, mp2 );
       }
       return FALSE;

    }  /* end FinanceFileWndProc() */
```

E.2.3 Source Code for the PWFin.MAK file

```
#*****************************************************************************
#  Dot directive definition area (usually just suffixes)
#*****************************************************************************

.SUFFIXES: .c .obj .dll .csc .sc .h .ih .ph .psc .rc .res

#*****************************************************************************
#  Environment Setup for the component(s).
#*****************************************************************************

SOMTEMP = .\somtemp
SCPATH  = D:\toolkt20\sc
HPATH   = D:\toolkt20\c\os2h
LIBPATH = D:\toolkt20\os2lib

!if [set SMINCLUDE=.;$(SCPATH);] || \
    [set SMTMP=$(SOMTEMP)] || \
    [set SMEMIT=ih;h;ph;psc;sc;c;def]
!endif

!if [cd $(SOMTEMP)]
!  if [md $(SOMTEMP)]
!    error Error creating $(SOMTEMP) directory
!  endif
!else
!  if [cd ..]
!    error Error could not cd .. from $(SOMTEMP) directory
!  endif
!endif

#
# Compiler/tools Macros
#

CC         = icc /c /Ge- /Gd- /Se /Re /ss /Ms /Gm+ /L+ /Li+ /Ls+ /Lx+ /La+
LINK       = link386
LDFLAGS    = /noi /map /nol /nod /exepack /packcode /packdata /align:16 /information
LIBS       = som.lib os2386.lib dde4mbs.lib

#*****************************************************************************
# Set up Macros that will contain all the different dependencies for the
# executables and dlls etc. that are generated.
#*****************************************************************************

OBJS       = pwFin.obj

#*****************************************************************************
#   Setup the inference rules for compiling source code to
#   object code.
#*****************************************************************************

.csc.c:
        sc -r $<

.c.obj:
        $(CC) -I$(HPATH) -c $<

all: pwFin.dll

pwFin.ih: $*.csc

pwFin.obj: $*.c $*.ih $*.h  $*.sc

pwFin.dll: $(OBJS) pwFin.res
         $(LINK) $(LDFLAGS) $(OBJS),$@,,$(LIBS),$*;
         rc $*.res $*.dll
         mapsym pwFin.map

pwFin.res: pwFin.rc
         rc -r $*.rc $*.res
```

E.2.4 Source Code for the PWFin.RC file

```
#define INCL_WIN
#include <os2.h>

#include "dialog.h"

ICON  ID_LOCK     LOCKED.ICO
ICON  ID_UNLOCK   UNLOCKED.ICO

MENU ID_CXTMENU_LOCK LOADONCALL MOVEABLE DISCARDABLE
BEGIN
   MENUITEM "~Lock Finance File", IDM_LOCK
END

MENU ID_OPENFinanceFile LOADONCALL MOVEABLE DISCARDABLE
BEGIN
   MENUITEM "Open Finance File", IDM_OPENFinanceFile
END*/


DLGTEMPLATE ID_DLG_PASSWORD LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG  "Password", ID_DLG_PASSWORD, 35, 26, 224, 76, FS_SYSMODAL |
            FS_SCREENALIGN | WS_VISIBLE, FCF_TITLEBAR
    BEGIN
        ENTRYFIELD      "", ID_EF_PASSWORD, 13, 38, 97, 8, NOT ES_AUTOSCROLL |
                        ES_MARGIN | ES_UNREADABLE
        LTEXT           "Finance File is locked.  Please enter password to open",
                        102, 10, 56, 197, 8
        DEFPUSHBUTTON   "Ok", DID_OK, 36, 9, 40, 14
        PUSHBUTTON      "Cancel", DID_CANCEL, 137, 10, 40, 14
    END
END
```

E.2.5 Source Code for the Dialog.H file

```
#define ID_DLG_PASSWORD            100
#define ID_EF_PASSWORD             103

#define ID_DLG_FIND                200
#define ID_EF_TELNUMBER            201
#define ID_EF_SURNAME              203

#define ID_LOCK                    300
#define ID_UNLOCK                  301

#define ID_CXTMENU_LOCK            0x6501
#define IDM_LOCK                   0x6502
#define IDM_QUERY                  0x6503

/*
 *   The following user-defined Popup menu items (ID_xxx) should be higher
 *   than WPMENUID_USER.
 *
 *   The ID_OPENFinanceFile will become a submenu of the system's popup open menu
 *   id, WPMENUID_OPEN.
 *
 *   These menu ids don't really matter.  The MENU could be any number.
 *   It is the MENUITEMS that have to be greater WPMENUID+*
 */
#define ID_OPENFinanceFile                  0x6504
#define IDM_OPENFinanceFile                 0x6505

/*   Set unique view ids. */
/*   define OPEN_ constant to represent the new view.  */
/*   *Must* be equal to the menu id used for this view.*/
#define OPEN_FinanceFile           IDM_OPENFinanceFile


#define ID_FRAME          3000                    /* client window id */
```

Glossary

ACDI. Asynchronous Communication Device Interface; programming interface which supports device-independent communication between devices on an asynchronous communications link. ACDI is supported by OS/2 Extended Edition Version 1.x, OS/2 Extended Services, and by a number of other hardware and software vendors.

action. A user-specified operation that is applied to an object.

action window. Term used in SAA CUA'91 documentation to refer to a short-lived window used to display information and receive input from the user in a structured dialog format. The information is typically related to a particular action being performed by the application. Implemented in Presentation Manager using a dialog box.

address conversion. The process of converting a 0:32 memory reference to a 16:16 memory reference, and vice versa. Part of thunking.

address translation. **(1)** The process of resolving a 0:32 memory reference into a physical memory address. When using the paged memory option in the 80386 processor, a memory pointer passed by an application consists of Page Directory and Page Table entries, and an offset within a physical page. This is resolved by the processor into a 32-bit physical memory address. The validity and legality of the memory reference is also checked during the translation process, and a general protection exception is generated if necessary. **(2)** The process of resolving a 16:16 memory reference into a physical memory address using a process's local descriptor table. The validity and legality of the memory reference is also checked during the translation process, and a general protection exception is generated if necessary.

ancestor. Term used in Workplace Shell programming to refer to any object class from which a new object class inherits methods and/or data. An ancestor may be the parent of the new object class, a parent of the parent, etc. See also descendant.

API. Application Programming Interface; term used to described the set of functions provided by which an application may gain access to operating system services.

APPC. Advanced Program-to-Program Communication; programming interface for peer-level communication between applications over an SNA LU6.2 communications link. APPC is supported in the PWS under DOS (using the APPC/PC product), and under OS/2 Extended Edition Version 1.x and OS/2 Extended Services using the Communications Manager. It is also supported in AS/400 systems and in System/370 hosts running VM/CMS and CICS.

application-controlled viewport. Viewport within a help window or online document window, where the display of information within that viewport is controlled by an application, which is specified by the developer of the source information. Application-controlled viewports may be used to display image, video or other types of information under the control of the Information Presentation Facility. See also IPF-controlled viewport.

application-modal. Term used to describe a message box or dialog box for which processing must be completed before further interaction with any other window owned by the same application may take place. Interaction with windows owned by other applications is unaffected.

application object. An object consisting of a particular representation of a data entity, along with its associated methods; the term is used in this document for clarity, in order to differentiate the concept from that of a data object.

application resource. Term used to describe a development resource that is application-specific. Such resources include source code modules and dynamic link libraries that will be used by only a single application.

ASCII. American Standard Code for Information Interchange; system which defines a standard for the representation of alphanumeric information within computer systems.

asynchronous processing. Invocation of another procedure whereby that procedure is dispatched as a separate thread of execution, independent of the caller. Control returns to the caller as soon as dispatching is complete; the caller may then continue execution without waiting for the dispatched procedure to complete its processing. Asynchronous processing enables a Presentation Manager application to preserve the user responsiveness goals laid down by Common User Access guidelines, by executing lengthy operations as separate threads of execution.

baselining. Process of establishing that a user-level application resource passes defined unit testing criteria before being promoted to production level. See also promotion and production-level.

base storage class. Term used to describe the three System Object Model object classes that form the basis of the inheritance hierarchy implemented by the Workplace Shell.

bit. A binary digit, which may have a value of either zero or one. Bits are represented within a computing device by the presence or absence of an electrical or magnetic pulse at a particular point, indicating a one or zero respectively.

byte. A logical data unit composed on eight binary digits (bits).

CASE. Computer Aided Software Engineering; term used to describe the concept of automated application design and code generation, based upon a set of requirements entered by a user. Products that implement CASE concepts are known as CASE tools.

cdecl. Linkage convention used in 16-bit "C" language programming under the IBM C/2 compiler, which causes the compiler to generate object code for a function or subroutine, such that parameters are placed on the stack in right-to-left order when the subroutine is called, and the calling routine clears the stack after control is returned. This is the default for "C" programs. Contrast with *pascal* linkage convention.

checkout. Process of recording the drawdown of an application resource by a developer for modification.

child window. A window that resides wholly within another window, known as its parent window, and that is restricted to the boundaries of the parent window.

class-based. Term used to describe an implementation of object-oriented programming, whereby object classes are defined in terms of other, previously-defined object classes; the new class takes on the properties and methods of the existing class or classes, in accordance with the principle of inheritance. The various dependencies thus formed constitute an inheritance hierarchy. Such an approach facilitates the creation of new object classes with minimal effort, but at the expense of increased interdependence and reduced object granularity. Compare with module-based approach.

class data. Data item for which the definition and contents are determined for an entire object class rather than an individual instance of that class; contrast with instance data.

class definition file. ASCII file that defines the characteristics of a Workplace Shell object class; used as input to the SOM Precompiler.

client area. See client window.

client-server. Term used to describe an application architecture whereby an application module (the server) accepts requests from a number of other application modules (the clients) and performs actions on behalf of these clients. Presentation Manager applications may be designed in such a way as to embody a client-server architecture.

client window. The area within a window that is used by an application to display information. The client window is actually a child window of the frame window within which it resides.

Common Programming Interface. Component of Systems Application Architecture that defines a set of programming languages and interfaces.

Common User Access. Common User Access; component of Systems Application Architecture that comprises a set of guidelines for defining user interface elements of an application. CUA includes such items as screen layout, PF key usage, etc.

compatability region. In the OS/2 Version 2.0 flat memory model, the address region below 512 MB, which may be addressed by a 16-bit application using the 16:16 addressing scheme. Under OS/2 Version 2.0, this region is equivalent in size to the process address space.

compatability region mapping algorithm. Algorithm used to perform address translation between 16:16 and 0:32 addressing schemes. See also thunking.

complex viewport. Term used to describe the creation of multiple viewports within the same help window or online document window, with separate formatting and scrolling characteristics. For example, one viewport may contain a bitmap, while another contains supporting text. The text viewport may be scrolled, while the other viewport containing the bitmap remains constant in the window. See also simple viewport.

context switching. In a multitasking operating system, the act of halting a currently executing task, saving its task state, and loading and dispatching a new task. Note that OS/2 undertakes context switching on the basis of threads rather than hardware-defined tasks.

control. See control window.

Control Panel. Application provided by the OS/2 Presentation Manager user shell, which allows a user to set and customize various system parameters and attributes such as screen colors, default printer settings, etc.

control window. A specialized window created for a specific purpose, such as a text entry field, a push button or a list box. The general concept of control windows also includes children of the frame window such as the title bar, menu bar and system, minimize and maximize icons. Also known simply as a control.

cooperative multitasking. Multitasking implementation whereby applications executing in the system must voluntarily relinquish control of the processor in order to allow another task to execute.

Cooperative multitasking is typically less reliable than pre-emptive multitasking, and results in lower system throughput.

CPI. See Common Programming Interface.

CRMA. See compatability region mapping algorithm.

CUA. See Common User Access.

DASD. Direct Access Storage Device; in traditional personal computer parlance, a fixed disk drive.

data object. A specific representation of a logical data entity; for example, the same logical data entity may exist both in memory and on a disk file. These representations are considered to be distinct data objects. See also application object.

descendant. Term used in Workplace Shell programming to denote any object class which inherits methods and/or data from the current object class. See also ancestor.

desktop. The screen background in the OS/2 Presentation Manager environment, upon which windows are displayed.

development resource. Identifiable component of an application, such as a source code module, dynamic link library, Presentation Manager icon, etc; alternatively, a utility or tool used to develop an application, such as a compiler, link-editor or programmers' toolkit.

dialog. An interaction between an application and a user, based upon the need to achieve a single logical task, such as the opening of a file.

dialog box. A special, short-lived window created by an OS/2 Presentation Manager application to display information and receive input from the user in a structured dialog format. The information is typically related to a particular action being performed by the application.

Dialog Box Editor. Utility application provided with the *IBM Developer's Toolkit for OS/2 2.0*, which allows the design and creation of templates for dialog boxes.

dialog procedure. Special case of a window procedure; associated with a dialog box rather than a standard window.

direct manipulation. User interface technique whereby application functions are initiated by the user manipulating objects, represented by icons, on the Presentation Manager or Workplace Shell desktop. The user typically initiates an action by selecting an icon, pressing and holding down a mouse button while "dragging" the icon over another object's icon on the desktop. The user then "drops" the icon over the target object by releasing the mouse button. For this reason, the technique is also known as "drag and drop" manipulation.

DLL. See dynamic link library.

domain. Logical grouping of users, applications and devices in a network, which is treated as a coherent unit for network administration purposes.

dormant. Term used to denote the state of a Workplace Shell object when no views of that object are currently open. Dormant objects typically do not "own" system resources, and their instance data is in an unknown state.

DOS. Disk Operating System; operating system for programmable workstations, originally developed by Microsoft Corporation as MS DOS, and marketed under license by IBM. Later versions were developed jointly by Microsoft and IBM.

DOS Compatibility Box. Facility provided by OS/2 Version 1.3 that allows a single real mode (DOS) application to execute under control of OS/2, occupying the lowest 640 KB of physical memory. A real mode application executing in this way may coexist with other protected mode applications in the same system. Certain restrictions are imposed on applications executing in the DOS Compatibility Box; an application may not be a TSR application, nor should it be a time-dependent application (such as a terminal emulator). These limitations are overcome by the Multiple Virtual DOS Machines feature implemented by OS/2 Version 2.0.

DOS settings. Feature implemented within the Multiple Virtual DOS Machines component of OS/2 Version 2.0, enabling a virtual DOS machine to be customized to suit the requirements of an application running within it. This feature enables greater application compatibility for virtual DOS machines.

drag and drop. See direct manipulation.

dragitem. In the context of direct manipulation, an item that is currently being dragged on the desktop by the user.

drawdown. Process of copying an application resource from a master-level library to a developer's own working library.

dynamic linking. Process under OS/2, whereby resolution of external references within an application to modules in dynamic link libraries (DLLs) is deferred until load time or run time. This allows modification of modules contained in DLLs without the need to link-edit an application once more. Since DLLs are normally written as reentrant code, dynamic linking also allows multiple applications to use the same copy of a DLL in memory, thus reducing the storage requirements of OS/2 applications.

dynamic link library. Library containing OS/2 application code and/or resources, which may be linked by one or more applications using the OS/2 dynamic linking process.

EHLLAPI. Emulator High-Level Language Application Programming Interface; programming interface provided by OS/2 Extended Edition Version 1.x and OS/2 Extended Services, which enables PWS applications to emulate keystroking for host terminal emulation sessions.

encapsulation. Concept whereby the definition of a data object is included as part of the application object that owns and manipulates the data object.

entity. Data item that must be manipulated by an application. See also logical data entity, data object and application object.

event-driven. Term used to describe an application environment where the sequence of processing is determined primarily by the application's response to external events, which may be initiated by the user, the operating system or other applications.

event semaphore. Semaphore used by applications to signal an event such as the termination of a thread.

exception handler. Application-supplied routine that is registered with OS/2 Version 2.0 on a per-thread basis, and invoked by the operating system whenever an exception condition occurs for that thread. Multiple exception handlers may be chained for each thread, and a handler may process an exception or allow it to pass on down the chain. OS/2 Version 2.0 itself provides default exception handlers for all exception conditions.

execution instance. An individual process executing a reentrant code module. Such a module may have multiple execution instances in existence at any one time; under OS/2, the operating system maintains a separate task state for each execution instance.

exportable entry point. Entry point for an application function that is invoked from outside the current executable module; for example, window procedures, which are invoked by Presentation Manager on the application's behalf, rather than by the application itself.

external entity. An entity such as a remote system or data entry device which exists outside the application, and interacts with the application. Such an entity and the methods to manipulate it may be encapsulated in an application object.

external reference. Program call to a memory address outside the current executable module; the target of the call is typically a library routine. Examples of external references are OS/2 system service calls.

EXPORTS statement. Statement used in a module definition file to define exportable entry points. Required by all OS/2 Presentation Manager applications under OS/2 Version 1.3 unless the *-Alfu* option is specified at compile time.

far call. When using the segmented memory model, a program call to a routine located in a different memory segment; such a call must use both a segment address and an offset in the branch instruction, rather than just the offset as is the case with a near call. By default, C programs compiled with the medium or large memory models generate far calls.

FIFO. First In, First Out; term used to describe a queueing algorithm whereby items are serially inserted into and removed from a queue structure, and where the first item placed into a queue structure is the first item to be removed.

flat memory model. Conceptual view of real memory implemented by OS/2 Version 2.0, where the operating system regards memory as a single linear address range of up to 4 GB.

focus. See input focus.

Font Editor. Utility provided with the *IBM Developer's Toolkit for OS/2 2.0*. which enables the design and creation of new fonts.

frame area. See frame window.

frame window. A window that contains the border visible on the screen; various control windows such as the title bar, menu bar and icons are children of the frame window. The client window, used by the application to display information, is also a child of the frame window.

functional decomposition. Application design technique also known as structured programming, whereby procedures become the focus of the application design and are progressively divided into smaller units of work until an atomic level is reached that may be implemented by a series of operations in a conventional programming language.

GB. Gigabyte; 1024 Megabytes, or 1024 x 1024 x 1024 bytes.

graphical model. Object-action, event-driven user interface model defined in the *IBM Systems Application Architecture CUA Advanced Guide to User Interface Design*. The graphical model makes extensive use of windows and graphical visual cues, and is supported by Presentation Manager.

GRE. Graphical Rendering Engine; component of the OS/2 Version 2.0 operating system responsible for drawing graphical items on the display.

guard page. Uppermost committed page of a private memory object, or lowermost committed page of a shared memory object, set using the PAG_GUARD flag. A write operation into a guard page generates a guard page exception, which may be trapped by an application-registered exception handler, which may then commit additional pages of storage. See exception handler.

guard page exception. Exception condition generated by OS/2 Version 2.0 when an application writes into a guard page. Like other exception conditions, the guard page exception may be trapped by an exception handler registered by the application. See exception handler.

help instance. A link between an application and the Information Presentation Facility, which allows information to be passed between the two, in order to display help panels and enable the application to be informed of events occurring within help windows.

help library. A file produced by the IPF compiler, which contains a collection of help panels related to a particular application, and which is used by the Information Presentation Facility to display help information for that application.

help panel. A unit of help information, displayed in a help window under the control of the Information Presentation Facility.

help window. A window displayed by the Information Presentation Facility at the request of an application, containing a help panel with information pertaining to the active application window.

Hungarian notation. Technique for including data typing in variable and procedure names within application source code; devised by Charles Simonyi of Microsoft.

hypergraphics. Under the Information Presentation Facility, a portion of a bitmap displayed in a help panel or online document, which may be selected by the end user. Selecting such an item causes an event to occur, such as the display of another help panel, a popup window, the dispatch of a message to the parent application or the start of a new application. See also hypertext.

hypertext. Under the Information Presentation Facility, a word or phrase in a help panel or online document, which may be selected by the end user. Selecting such an item causes an event to occur, such as the display of another help panel, a popup window, the dispatch of a message to the parent application or the start of a new application. See also hypergraphics.

icon. In the Presentation Manager user interface, a small graphical image that represents an application, window or other object such as a file or device. The user may directly manipulate icons on the desktop to perform work tasks.

Icon Editor. Utility application provided with OS/2 Version 2.0, which may be used to design and create icons.

import library. Library file containing information on functions and resources contained in a dynamic link library. Used as input to the OS/2 linkage editor. Use of an import library negates the need for an IMPORTS statement in an application's module definition file.

IMPORTS statement. Statement used in a module definition file to denote functions that will be imported from a dynamic link library. Required for any application that uses a DLL, unless an import library containing information on the DLL functions is specified at link-edit time.

Information Presentation Facility. Facility provided by OS/2 Version 1.2 and above (including OS/2 Version 2.0), by which application developers may produce online documentation and context-sensitive online help panels for their applications, using an IPF tag language and an IPF compiler.

inheritance. Concept whereby an application object takes on the properties of the object class to which it belongs; such properties may include the definition of instance variables and data objects. Object classes may also be defined in terms of other, previously defined object classes; the new object class then takes on the properties and methods of its parent class or classes (which themselves may be defined in terms of other classes).

inheritance hierarchy. Term used to describe the chain of interdependence formed when object classes are defined in terms of other currently existing classes, and where the definition of an object class is thus dependent upon the presence of its parent class or classes.

input focus. In the Presentation Manager environment, the user interacts with a single window at any given moment; that window is said to have the input focus.

instance. An individual application object (or Workplace Shell object) that belongs to a particular object class. See also execution instance.

instance data. A data item that is created and owned by a single instance of an application object class. An instance variable is typically defined by the class, and the definition is thus common to all instances of that class, but a separate data item is maintained for each instance.

Interface Control Document. Document that contains descriptions of all generic library objects and functions, along with their external interfaces including both input and output data. The interface control document is used by application developers who wish to utilize library objects and functions in their applications.

IPF. See Information Presentation Facility.

IPF compiler. Compiler provided with the *IBM Developer's Toolkit for OS/2 2.0*, which allows the compilation of source files into help libraries or online documents that may be displayed using the Information Presentation Facility.

IPF-controlled viewport. Viewport within a help window or online document window, where the formatting and display of information within that window is controlled by the Information Presentation Facility. This is the default case for information displayed using the Information Presentation Facility. See also application-controlled viewport.

IPF tag language. Text formatting language used by the Information Presentation Facility, whereby tags are inserted into the ASCII source text to cause formatting to occur. The tags and syntax of the IPF tag language are similar to IBM's Generalized Markup Language (GML).

KB. Kilobyte; 1024 bytes.

LAN. See local area network.

LIFO. Last In, First Out; term used to describe a queueing algorithm whereby items are serially inserted into and removed from the queue structure, and where the most recently inserted item is the first item to be removed.

load-time dynamic linking. Form of dynamic linking where resolution of external references takes place during loading of the application into memory. Load-time dynamic linking is typically used to share a single memory-resident copy of common application service routines, or to isolate portions of application code from the remainder of the application, thereby facilitating any future change management. See also run-time dynamic linking.

local area network. A network, typically composed of programmable workstations, in which the constituent nodes are situated in direct geographical proximity to one another, usually within the same property boundary. A number of local area networks may be "bridged" together to form a wide area network that may extend across multiple geographical locations.

logical data entity. Term used to describe a coherent unit of logical data, such as a block of text. This data may simultaneously exist in numerous representations within the system, either in memory or on secondary storage devices. See also data object and application object.

MB. Megabyte; 1024 kilobytes, or 1024 x 1024 bytes.

memory object. Logical unit of memory requested by an application, which forms the granular unit of memory manipulation from the application viewpoint. A memory object may be up to 512 MB in size under OS/2 Version 2.0.

menu bar. Area near the top of a CUA-conforming window, which contains a horizontally oriented list of actions that relate to the data object being manipulated in the window.

message. Data structure used by OS/2 Presentation Manager for communication between window procedures, or between Presentation Manager and a window procedure. Messages may be system-defined or user-defined. See also message class.

message box. Within Presentation Manager, a specialized type of dialog box that carries out a simple structured dialog with the user. This dialog is limited to the display of some textual information, and the input of a single choice from a limited, finite set of mutually exclusive options. A message box is simpler to process than a dialog box, and is useful in circumstances where the sophistication of a dialog box is unnecessary.

message class. Logical grouping of messages with common properties. Under Presentation Manager, messages are grouped into classes with the name of the message class and the definitions of message parameters being common to all instances of that class. Individual instances differ in their destination window and in the contents of their message parameters.

metaclass. Term used to describe the conjunction of an object and its class information; that is, the information pertaining to the class as a whole, rather than to a single instance of the class. Each class is itself an object, which is an instance of the metaclass.

method. In object-oriented terms, a routine or procedure used to manipulate a data object. A group of methods, together with a definition of the data object itself, constitute an application object.

mixed model programming. Technique of combining 16-bit and 32-bit modules and resources within the same application under OS/2 Version 2.0. Mixed model programming requires some special considerations when passing control and parameters between 16-bit and 32-bit modules and resources.

modal. Term used to describe a message box or dialog box for which processing must be completed before user interaction may continue. Two types of

modal dialog are possible: application-modal and system-modal. See also modeless.

modeless. Term used to describe a dialog box which is displayed on the screen, but which does not require immediate interaction with the user, who may continue to interact with other windows while the dialog box is displayed.

module-based. Term used to describe an implementation of object-oriented programming, whereby an object class is completely defined in its own right, and does not depend upon the definitions of other classes. Such an approach provides increased levels of granularity and thus enhances object reusability, albeit at the expense of additional complexity during creation of the object. Compare with class-based approach.

module definition file. ASCII text file typically used with OS/2 Presentation Manager applications to define information such as module names, stack and heap sizes and exportable entry points. Used as input to the OS/2 link-editor.

Multiple Virtual DOS Machines. Feature of OS/2 Version 2.0 that enables multiple DOS applications to execute concurrently in full-screen or windowed mode under OS/2 Version 2.0, in conjunction with other 16-bit or 32-bit applications, with full pre-emptive multitasking and memory protection between tasks.

multiprogramming. Term used to describe an operating system environment that allows the concurrent execution of multiple applications in the same system.

multitasking. Extension of the multiprogramming concept, whereby processor time is distributed among multiple applications by giving each application access to the processor for short periods of time. Multitasking may be cooperative or pre-emptive in nature.

multithreading. Extension of the multitasking concept, whereby multiple "tasks" may exist within a single application, allowing different portions of the application to execute asynchronously to one another. Multithreading allows an application to continue to respond to user input while lengthy processing takes place in the background.

Mutex semaphore. Semaphore used to control access to a resource that may be safely accessed only by a single application thread at any time.

MuxWait semaphore. Semaphore used when waiting for multiple events to occur or for multiple resources to be freed.

near call. Under the segmented memory model, a program call to a routine located within the same memory segment. Such a call provides only the offset of the routine within the segment; a segment address is not needed since all application code is assumed to reside in the same memory segment. By default, C programs compiled using the small or compact memory models generate near calls. See also far call.

NPX. Numeric Processor Extension; term used in reference to the exception condition generated by the 80386 processor when an application issues a numeric coprocessor instruction in a machine with no coprocessor installed. Note that OS/2 Version 2.0 will trap the NPX exception and emulate the numeric coprocessor function within the operating system, returning the result to the application exactly as if a physical coprocessor were installed.

object. A collection of data and methods (procedures) that operate on that data, which together represent a logical entity in the system. In object-oriented programming, objects are grouped into classes that share common data definitions and methods. Each object in the class is said to be an instance of the class. See also instance, object class, data object, application object.

object class. Logical grouping of application objects with similar properties; methods are normally associated with an object class, rather than with individual instances of that class. Under the Workplace Shell, an object class is defined using a class definition file.

object window. A conceptual window that does not appear on the screen, and is associated with a data object such as a file or database. In effect, an object window is merely an address to which messages may be sent to cause manipulation of a particular data object.

online document. A set of information relating to an application or a business process, which is developed using the IPF tag language, compiled and viewed using the Information Presentation Facility. An online document, unlike a help library, is not necessarily related to a single application, and exists in a "standalone" environment.

online document window. A window displayed by the Information Presentation Facility, using the VIEW.EXE utility, and which contains information from an online document.

Operating System/2. Operating system for programmable workstations, offering more sophisticated multiprogramming and multitasking capabilities and larger memory support than DOS.

optimized window. Term used to describe a dialog box; such a window is created for a specific purpose, containing a number of control windows, and is deemed to have been created at the optimum size to display those control windows. Thus an optimized

window may not be resized by the user, nor may it be minimized or maximized.

optlink. Default linkage convention using in C programming under the IBM C Set/2 compiler, which causes the compiler to generate object code for a function or subroutine, such that parameters are placed on the stack in right-to-left order, and the stack is cleared by the calling function when control is returned. The *optlink* linkage convention differs from the *system* linkage convention in the preservation of register values and the handling of return values.

OS/2. See Operating System/2.

page. Granular unit for memory management using the 80386 or 80486 processors with the paging feature enabled. A page is a 4 KB contiguous unit of memory, which the processor manipulates as a single entity for the purpose of memory manipulation and swapping.

parent. Immediate ancestor of a Workplace Shell object class; the object class from which the current object class was created by subclassing. See also ancestor, descendant.

pascal. Linkage convention used in C programming that causes the compiler to generate object code for a function or subroutine, such that parameters are placed on the stack in left-to-right order when the subroutine is called, and the stack is cleared by the called subroutine when control is returned. Originally introduced by Microsoft with early versions of Microsoft Windows, when this convention saved several hundred bytes of system memory, and adopted by Presentation Manager under OS/2 Version 1.3. Contrast with *cdecl* linkage convention.

PC DOS. Personal Computer Disk Operating System; see DOS.

persistent. Term used to describe Workplace Shell objects that are registered with the operating system and that continue to exist in the Workplace Shell after the system is re-IPLed.

pipe. Facility provided by OS/2 for communication between threads and/or processes. Pipes may be named or unnamed.

polymorphism. Concept whereby the behavior of an application object is dependent solely upon the class and contents of the messages received by that object, and is not affected by any other external factor. Similarly, the effect of the same message class is dependent upon the application object to which it is passed, and that object's interpretation of the message.

pre-emptive multitasking. Multitasking implementation whereby the operating system provides a scheduler that periodically interrupts an executing task, saves its current state information and dispatches another task, thereby sharing system resources across multiple applications.

Presentation Interface. Component of Systems Application Architecture Common Programming Interface, which defines an application programming interface for presentation services.

Presentation Manager. Windowed, graphics-oriented user interface provided by OS/2 Version 1.1 and above; allows object-oriented applications to be written in conformance with Common User Access guidelines. The Presentation Manager programming interface conforms to the Systems Application Architecture Common Programming Interface Presentation Interface specifications.

presentation space. In the context of Presentation Manager, a conceptual display surface onto which information is written. A presentation space is a data object normally associated with a window.

primary thread. The first thread of execution created by an OS/2 application upon application startup. In a Presentation Manager application, the primary thread typically provides processing for user interaction and short-lived application functions. Lengthy processing is typically carried out by a secondary thread, to preserve user responsiveness goals as defined in Common User Access guidelines.

private semaphore. See semaphore.

procedural entity. An entity, such as an administrative procedure, that interacts with other logical entities within an object-oriented application, and that itself may be encapsulated within an application object.

process. In the context of OS/2, an instance of a particular program, which owns system resources such as threads, file handles and a memory map described by the process' local descriptor table. A process may consist of one or more threads.

process address space. Region of memory addressable by a single process under OS/2 Version 2.0. Each process address space may be up to 512 MB in size.

production-level. Term used to describe an application resource that has been unit-tested and baselined, and is placed in a production library from which multiple application developers may access it.

programmable workstation. An intelligent workstation device, containing its own processor, memory and possibly its own storage devices; for example, the IBM Personal System/2 family.

promotion. Process of migrating a user-level application resource to production level after baselining.

protect mode. Native mode of execution for the Intel 80286 processor, whereby memory addresses are made by segment and offset, and a separate set of descriptors is maintained for memory addressable by each application, thereby providing memory isolation in a multiprogramming environment.

PTDA. Per Task Data Area; storage area maintained by OS/2 for each process executing in the system, containing process-specific execution data.

pulldown menu. A menu that appears when the user selects an item in a window's menu bar. The pulldown menu acts as a "submenu," and contains a list of entries from which the user may select the required action. Pulldown menus may be nested.

PWS. See programmable workstation.

queue. Facility provided by OS/2 for communication between threads and/or processes.

real mode. Mode of execution for the Intel 80286 processor, whereby the processor emulates the operation of an 8086/8088 processor. In this mode, memory addresses are made to physical locations in memory, and addressability is limited to 1 MB. Real mode is the mode used by the DOS operating system; the design of this operating system further restricts applications' memory addressability to 640 KB.

rendering mechanism. Protocol that defines the communication between two windows or objects during a drag/drop operation. Presentation Manager provides three standard rendering mechanisms for commonly used data types, and application developers may define their own rendering mechanisms to meet specific requirements.

resource. Item such as a window or dialog template, string table, menu table, etc., which may be defined externally and used by one or more Presentation Manager or Workplace Shell applications.

resource compiler. Utility provided with *IBM Developer's Toolkit for OS/2 2.0*, which takes a resource script file and produces a precompiled resource or group resources that may be incorporated into an executable code module.

resource script file. ASCII text file with .RC extension, used to define resources for a Presentation Manager application. Used as input to resource compiler.

REXX. Restructured Extended Executor; procedural language included as part of OS/2 Version 2.0, which provides batch language functions along with structured programming constructs such as loops, conditional testing and subroutines. Programs or subroutines written using the REXX language may be invoked directly from the command line, or from an application written in REXX or another programming language.

run-time dynamic linking. Form of dynamic linking whereby modules and resources are loaded into memory during application execution, by the application issuing a **WinLoadModule()** function call. Run-time dynamic linking is typically used for portions of the application code that are rarely required (such as fatal error handling routines); they are therefore explicitly loaded into memory when needed. In this way, the memory requirements of the application may be reduced.

SAA. See Systems Application Architecture.

scheduler. Component of OS/2 which provides automated task dispatching.

secondary thread. Separate thread of execution created by a Presentation Manager application to handle lengthy processing of a specific type of message using an object window.

segment. Unit of memory addressable by the Intel 80x86 processors. With the 8086 and 80286 processors, a segment may be from 16 bytes to 64 KB in size. With the 80386 and 80486 processors, a segment may be up to 4 GB in size.

segmented memory model. Mode of addressing used by Intel 80x86 processors, whereby memory is addressed in segments. Individual units of up to 64 KB (8086/80286) or 4 GB (80386/80486) in size may be allocated by an operating system that uses this memory model.

seg16. #pragma directive used in C language programming under the IBM C Set/2 compiler, which ensures that automatic data structures do not cross a 64 KB segment boundary, and are thus addressable by both 16-bit and 32-bit code. This directive is used when declaring data structures that will be used as parameters when invoking 16-bit functions or subroutines from 32-bit applications.

semaphore. Data structure provided by OS/2, and used for synchronization between threads and/or processes. OS/2 Version 2.0 allows mutex semaphores, event semaphores and muxwait semaphores. Semaphores may also be *private* (owned and accessible by a single process) or *shared* (accessible by all processes in the system). A process may create up to 65,535 private semaphores, and there may be a system-wide total of up to 65,535 shared semaphores in existence at any time.

shared semaphore. See semaphore.

siblings. Term used in the Presentation Manager environment to describe two or more windows that have the same parent window.

simple viewport. A viewport within a help window or online document window that is the only viewport within that window. This is the default case for the display of text or graphics using the Information Presentation Facility. See also complex viewport.

SNA. System Network Architecture; defined series of layered interfaces and protocols for communication between systems and devices.

SOM. See system object model.

SOM Precompiler. Precompiler which generates C source code and header files from a class definition file. The resulting code may then be edited by a programmer to add application logic, and then compiled using a normal C compiler such as C Set/2.

source. In a direct manipulation operation, the object or program from which a dragitem is being dragged. See also direct manipulation, dragitem, target.

sparse object. Under OS/2 Version 2.0, a memory object that has been allocated but for which no storage has yet been committed. A sparse object has a valid address range in the process address space, but cannot be the target of a write operation until one or more pages within the object are committed.

SQL. Structured Query Language; Systems Application Architecture-conforming language for the definition and manipulation of data stored in relational database management systems. SQL is supported in the programmable workstation by OS/2 Database Manager, in the AS/400 midrange systems and in System/370 hosts by the DB2 and SQL/DS products.

SRPI. Server-Requester Programming Interface; programming interface enabling master-slave communication between a workstation application (the requester) and a host application (the server) using an SNA LU2.0 communications link. SRPI is supported in the PWS under DOS, OS/2 Extended Edition V1.x and OS/2 Extended Services, and in System/370 hosts running VM/CMS and MVS/TSO.

structured programming. Application design technique whereby applications are successively broken down into their component functions until a level is achieved at which application code may easily be generated. See functional decomposition.

subclassing. Technique whereby messages destined for a particular object are diverted to another object that may perform special processing for a particular message type or provide additional methods not provided by the parent object. Subclassing is typically performed on individual instances of an object class, rather than on the entire class.

synchronous processing. Invocation of another procedure whereby control does not return to the caller until that procedure has completed its processing.

system. Linkage convention used in C programming under the IBM C Set/2 compiler, which causes the compiler to generate object code for a function or subroutine, such that parameters are placed on the stack in right-to-left order, and the stack is cleared by the calling function when control is returned. The *system* linkage convention differs from the default *optlink* linkage convention in the preservation of register values and the handling of return values. Use of the *system* linkage convention is required for functions that are invoked by the operating system or Presentation Manager, such as window and dialog procedures. The *system* linkage convention is specified using the #pragma linkage directive.

system-modal. Term used to describe a message box or dialog box for which processing must be completed before further interaction with any other window in the system may take place.

system object model. Set of base object classes and object-relationship protocol definitions that defines a basic object-oriented layer on top of Presentation Manager, and which is exploited by the Workplace Shell.

Systems Application Architecture. Set of guidelines for application development, covering areas such as user interfaces, application programming interfaces and communications environments. Systems Application Architecture facilitates ease of use and application portability between environments.

target. In a direct manipulation operation, the object or program to which a dragitem is being dragged. See also direct manipulation, dragitem, source.

task state. Set of information that embodies the current state of a thread (task) in the system; composed of the register contents and stack belonging to a thread at the time it is pre-empted by the scheduler.

thread. Atomic unit of dispatch under OS/2. One or more threads make up a process; the threads within a process may share resources belonging to that process.

thunking. Term used to describe the process of address conversion, stack and structure realignment, etc., necessary when passing control between 16-bit and 32-bit modules under OS/2 Version 2.0.

thunk procedure. Application procedure that performs thunking.

timer. Facility provided under OS/2 Presentation Manager, whereby Presentation Manager will dispatch a message of class WM_TIMER to a particular window at specified intervals. This capability may be used by an application to perform a specific processing task at predetermined intervals, without the necessity for the application to explicitly keep track of the passage of time.

title bar. Area at the top of a CUA-conforming window that contains the title of the window (typically identifying the data object or device to which the window relates).

transient object. A Workplace Shell object that does not persist beyond a system IPL. Transient object classes are descendants of the *WPTransient* object class.

TSR. Terminate-and-Stay-Resident; term used to describe a DOS application that modifies an operating system interrupt vector to point to its own location (known as hooking an interrupt). This process is not permitted under OS/2, although facilities exist under OS/2 to provide the same capability for applications.

user-level. Term used to describe an application resource that is currently undergoing modification and unit testing; when unit testing is complete, the resource is then submitted for baselining and subsequent promotion to production level.

view. In the Workplace Shell, a view is a window that displays the contents or properties of an object is a particular manner. For example, an "icon view" displays the contents of a container object as a series of icons. The same object may also have a "settings view," which displays the characteristics of the object.

viewport. Under the Information Presentation Facility, a portion of a help window or online document window that may be separately manipulated. The use of multiple viewports in a window enables the display of different types of information in the same window, with separate formatting and scrolling. Viewports may be either simple or complex, and may be IPF-controlled or application-controlled.

VIO. Term used to describe the OS/2 Video Subsystem, used by text-windowed and full-screen protect-mode applications executing under OS/2.

window. A conceptual identity under OS/2 presentation manager; a window is essentially a handle associated with a data object such as a presentation space on the screen or a database that, along with an associated window procedure, comprises an application object. Hence a window may be considered as equivalent to an application object.

window class. A group of windows having a common set of data object definitions and processing requirements, and which therefore share a common window procedure. In object-oriented terms, a window class equates to an object class.

window handle. Unique identifier of a window, generated by Presentation Manager when the window is created, and used by applications to direct messages to the window.

window procedure. A procedure in an OS/2 Presentation Manager application that processes messages intended for a particular window class. In object-oriented terms, a window procedure contains the data object definitions and methods associated with a particular application object.

window words. Storage area within the control block maintained for each window by Presentation Manager that is available for an application to store information such as a pointer to a memory object containing instance data.

winproc. See window procedure.

Workplace Environment. User interface model defined in the *1991 IBM Systems Application Architecture CUA Advanced Guide to User Interface Design*, whereby direct manipulation of icons is used to provide a conceptual analogy of the user's physical working environment.

Workplace Shell. Standard user interface component of OS/2 Version 2.0 that provides an object-oriented interface for the end user. The implementation of the Workplace Shell is based upon the system object model.

Workplace Shell object. An object created by the Workplace Shell, typically at the request of the user or an application. A Workplace Shell object is very similar in concept to an application object, in that it possesses data and methods that operate upon that data. See also application object.

z-order. Conceptual order of windows on the desktop; windows are considered to be located "one on top of another."

0:32. Term used to describe the addressing scheme used for the 32-bit flat memory model, where a memory address is expressed as a 32-bit linear offset within the linear address range.

16:16. Term used to describe the addressing scheme used for the 16-bit segmented memory model, where a memory address is expressed as a 16-bit segment selector, and a 16-bit offset within that segment.

80286. Intel 80286 microprocessor, 16-bit microprocessor that provides both real and virtual memory support, and allows multitasking and

automated task dispatching. Successor to the Intel 8086/8088 family.

80386. Intel 80386 microprocessor; the 32-bit processor upon which the OS/2 Version 2.0 operating system is based.

80486. Intel 80486 microprocessor; a 32-bit processor that implements a superset of the 80386 processor instruction set.

8086/8088. Microprocessor family developed by Intel Corporation for use in personal computers. The 8086/8088 family provides 8-bit real memory addressing mode only, and is limited to an address space of 1 MB.

_far16. Linkage convention used in C programming language within the #pragma linkage directive under the IBM C Set/2 compiler, to indicate that a function or subroutine resides in a 16-bit module, and that thunking is required when the function or subroutine is invoked.

_Seg16. Keyword used in C programming language under the IBM C Set/2 compiler, to indicate that a pointer should be stored in 16:16 format rather than 0:32 format. See also *seg16*.

#pragma linkage. Directive used with the IBM C Set/2 compiler to determine the linkage convention for a function or function type declaration. See also *optlink* and *system*.

#pragma seg16. Directive used with the IBM C Set/2 compiler to ensure that a data structure is aligned on a 64 KB segment boundary, and is thus addressable by 16-bit application code. See also *seg16*.

#pragma stack16. Directive used with the IBM C Set/2 compiler to set the stack size for all 16-bit function calls made from a 32-bit application module.

Index

Special Characters

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z

GG24-3774-01

OS/2 Version 2 Volume 4: Writing Applications

GG24-3774-01

PRINTED IN THE U.S.A.

IBM®

GG24-3774-01